

3 D O M 2 R E L E A S E ♦ V E R S I O N 2 . 0

Volume 4

- ♦ *3DO M2 Portfolio Programmer's Guide*
- ♦ *3DO M2 Portfolio Programmer's Reference*



3DO M2 Portfolio Programmer's Guide

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

Preface

About This Book	PPG-xxiii
Programmer's Guide vs. Programmer's Reference	PPG-xxiv
How This Book is Organized.....	PPG-xxiv
Related Documentation	PPG-xxv
About the Code Examples.....	PPG-xxvi
About Bit Ordering.....	PPG-xxvii
Typographical Conventions	PPG-xxvii

1

Understanding the Kernel

Description of the Kernel	PPG-1
Multitasking.....	PPG-2
Multitasking	PPG-2
Task Termination	PPG-4
Parent Tasks, Child Tasks, and Threads	PPG-4
Memory Management	PPG-5
Memory Size	PPG-5
Allocating Memory	PPG-5
Memory Fences	PPG-8
Returning Memory	PPG-9
Sharing Memory	PPG-9
Working with Folios.....	PPG-10
Managing Items.....	PPG-10
Items	PPG-10
Creating an Item	PPG-11
Opening Items	PPG-11

Using Items	PPG-11
Deleting Items	PPG-12
Semaphores	PPG-12
Intertask Communication.....	PPG-13
Signals	PPG-13
Messages	PPG-14
Portfolio Error Codes	PPG-16

2

Tasks and Threads

What Is a Task?	PPG-19
Starting and Ending Tasks	PPG-20
Creating a Task	PPG-20
Ending a Task	PPG-21
Controlling the State of a Task	PPG-22
Yielding the CPU to Another Task	PPG-22
Going To and From Wait State	PPG-22
Changing a Task's Parent	PPG-23
Changing Task Priorities	PPG-23
Task Quantums	PPG-24
Starting and Ending Threads	PPG-24
Creating a Thread	PPG-24
Ending a Thread	PPG-26
Advanced Task and Thread Usage	PPG-26
Example: Using Threads and Signals	PPG-27

3

Using Tag Arguments

About Tag Arguments	PPG-31
Special Tag Commands	PPG-32
How to Specify Tags	PPG-33
How Tags are Documented	PPG-33
How to Specify Tags Using a Tag Array	PPG-33
How to Specify Tags Using VarArg	PPG-34
Parsing Tags	PPG-35

4

Managing Linked Lists

About Linked Lists	PPG-38
--------------------------	--------

Characteristics of Portfolio Lists	PPG-38
Characteristics of Nodes	PPG-39
Creating and Initializing a List	PPG-39
Adding a Node to a List	PPG-40
Adding a Node to the Head of a List	PPG-40
Adding a Node to the Tail of a List	PPG-40
Adding a Node After Another Node in a List	PPG-40
Adding a Node Before Another Node in a List	PPG-41
Adding a Node According to Its Priority	PPG-41
Adding a Node in Alphabetical Order	PPG-41
Adding a Node According to Other Node Values	PPG-42
Changing the Priority of a Node	PPG-42
Removing a Node From a List	PPG-42
Removing the First Node	PPG-42
Removing the Last Node	PPG-43
Removing a Specific Node	PPG-43
Finding Out If a List Is Empty	PPG-43
Traversing a List	PPG-43
Traversing a List From Front to Back	PPG-43
Traversing a List From Back to Front	PPG-44
Finding a Node by Name	PPG-45
Finding a Node by Its Ordinal Position	PPG-45
Determining the Ordinal Position of a Node	PPG-46
Counting the Number of Nodes in a List	PPG-46
Primary Data Structures	PPG-46
The Node Structure	PPG-46
MinNode and NamelessNode Structures	PPG-47
The List Data Structure	PPG-48
The ListAnchor Union	PPG-48
The Link Data Structure	PPG-49

5 Managing Memory

About Memory	PPG-52
How Memory Works	PPG-52
Managing Memory	PPG-53
Allocating a Block of Memory	PPG-53
Freeing a Block of Memory	PPG-54
Reallocating a Block of Memory	PPG-54

Getting Information About Memory	PPG-55
Finding Out How Big a Block of Memory Is	PPG-55
Finding Out How Much Memory Is Available	PPG-55
Getting the Size of a Memory Page	PPG-56
Validating Memory Pointers	PPG-56
Allowing Other Tasks to Write to Your Memory.....	PPG-57
Transferring Memory to Other Tasks	PPG-58
Interfacing to the System's Free Page List Directly.....	PPG-58
Debugging Memory Usage	PPG-59
Example: Allocating and Deallocating Memory	PPG-61
Other Memory Topics (Caches, Bit Arrays).....	PPG-62
Caches	PPG-62
Bit Arrays	PPG-64

6

Managing Items

About Items	PPG-67
Creating or Opening an Item	PPG-69
Creating an Item	PPG-69
Specifying the Item Type	PPG-69
Tag Arguments	PPG-70
VarArgs Tag Arguments	PPG-71
The Item Number	PPG-72
Convenience Calls to Create Items	PPG-72
Using an Item	PPG-73
Finding an Item	PPG-73
Getting a Pointer to an Item	PPG-73
Checking If an Item Exists	PPG-74
Changing Item Ownership	PPG-74
Changing an Item's Priority	PPG-74
Deleting an Item	PPG-75
Opening an Item	PPG-75
Closing an Item.....	PPG-75

7

Semaphores

About Semaphores	PPG-77
The Problem: Sharing Resources Safely	PPG-78
The Solution: Locking Shared Resources When Using Them	PPG-78

The Implementation: Semaphores	PPG-78
Using Semaphores	PPG-78
Creating a Semaphore	PPG-79
Locking a Semaphore	PPG-79
Exclusive and Shared Access.....	PPG-80
Priority Inversion.....	PPG-80
Unlocking a Semaphore.....	PPG-80
Finding a Semaphore.....	PPG-81
Deleting a Semaphore.....	PPG-81

8

Communicating Among Tasks

About Intertask Communication	PPG-83
Using Signals.....	PPG-84
Allocating a Signal Bit	PPG-84
Receiving a Signal	PPG-85
Sampling and Changing the Current Signal Bits	PPG-86
Sending a Signal	PPG-86
Freeing Signal Bits	PPG-86
Sample Code for Signals	PPG-87
Passing Messages	PPG-87
Creating a Message Port	PPG-88
Creating a Message	PPG-89
Sending a Message	PPG-91
Receiving a Message	PPG-92
Working with a Message	PPG-93
Pulling Back a Message	PPG-94
Replying to a Message	PPG-94
Messages With No Reply Ports	PPG-95
Forwarding Messages to Another Port	PPG-96
Finding a Message Port	PPG-96
Example Code for Messages	PPG-96

9

The Portfolio I/O Model

Introduction.....	PPG-101
Hardware Devices and Connections.....	PPG-102
The Control Port	PPG-102
I/O Architecture	PPG-103

Devices	PPG-103
Device Stacks	PPG-104
Drivers	PPG-104
I/O Requests (IOReqs)	PPG-104
Performing I/O	PPG-105
Preparing for I/O	PPG-105
Creating an IOReq	PPG-106
Initializing an IOInfo Structure	PPG-107
Passing IOInfo Values to the Device	PPG-109
Asynchronous and Synchronous I/O	PPG-109
Completion Notification	PPG-111
Reading an IOReq	PPG-112
Continuing I/O	PPG-113
Aborting I/O	PPG-113
Finishing I/O	PPG-114

10

Portfolio Devices

Introduction	PPG-115
Timer Devices	PPG-117
Creating a Timer IOReq	PPG-117
The Microsecond Clock	PPG-117
The Vertical Blank Clock	PPG-120
File Devices	PPG-123
Important Note About Microcard File Systems	PPG-123
Methods for Communicating With File Devices	PPG-123
Creating, Deleting, Renaming, Getting Information about Files and Directories	PPG-124
Preparing to Send an IOReq to a File Device	PPG-124
Getting Filesystem Status	PPG-125
Getting File Status	PPG-126
Allocating Blocks for a File	PPG-127
Writing Blocks of Data to a File	PPG-127
Marking the End of a File	PPG-128
Reading Blocks of Data From a File	PPG-128
Setting the File Type	PPG-129
Setting the File Version and Revision	PPG-129
Setting the Virtual Block Size	PPG-129
Getting Directory Information	PPG-130
Cleaning Up After I/O Operations	PPG-131

Serial Devices	PPG-131
Writing Data to a Serial Port	PPG-131
Reading Data from a Serial Port	PPG-132
Configuring the Serial Port	PPG-133
Reading the Serial Port Configuration	PPG-134
Waiting for Particular Events	PPG-134
Controlling Serial Lines	PPG-135
Sending Break Signals	PPG-135
Getting Serial State	PPG-135

11

The File System, File Folio, and FSUtils Folio

The 3DO File System Interface.....	PPG-137
Files and File Functions	PPG-138
RawFiles and RawFile Functions	PPG-138
Directories and Directory Functions	PPG-138
Avatars	PPG-139
Pathnames and Filenames	PPG-140
File Functions.....	PPG-141
Creating and Deleting Files	PPG-141
Renaming Files and Directories	PPG-141
Opening and Closing Files (For Block-Oriented Operations)	PPG-141
Setting File Attributes	PPG-142
Finding Files	PPG-142
RawFile Functions.....	PPG-145
Opening a RawFile	PPG-146
Reading From a RawFile	PPG-147
Writing to a RawFile	PPG-147
Setting the Size of a RawFile	PPG-148
Seeking in a RawFile	PPG-148
Clearing Error Conditions for an Open RawFile	PPG-149
Getting Information About a RawFile	PPG-149
Setting the Attributes of a RawFile	PPG-150
Closing a RawFile	PPG-151
Directory Functions	PPG-151
Creating and Deleting Directories	PPG-151
Opening and Closing Directories	PPG-151
Finding and Changing the Current Directory	PPG-152
Reading a Directory	PPG-153

Mounting and Dismounting File Systems	PPG-153
Minimizing File Systems	PPG-153
Finding Mounted File Systems	PPG-154
File Folio Examples	PPG-154
Displaying Contents of a File	PPG-155
Scanning the File System	PPG-159
Listing the Contents of a Directory	PPG-162
The FSUtils Folio - File System Utilities.....	PPG-165
Copying a Directory Tree	PPG-165
Deleting a Directory Tree	PPG-168
Determining the Full Pathname of an Open File	PPG-169
Extracting the Final Component of a Pathname	PPG-169
Appending a Pathname to Another Pathname	PPG-169

12

The Batt Folio and Date Folio

Introduction - The Battery-Backed Clock	PPG-171
The Batt Folio.....	PPG-172
Reading the Current Gregorian Date and Time	PPG-172
Setting the Current Gregorian Date and Time	PPG-172
The Date Folio.....	PPG-173
Converting from Gregorian to TimeVal Representation	PPG-173
Converting from TimeVal to Gregorian Representation	PPG-173
Validating a Gregorian Date-Time Value	PPG-174

13

The Event Broker

About the Event Broker.....	PPG-176
Specifying and Monitoring Events	PPG-176
Working With Input Focus	PPG-177
Reconfiguring or Disconnecting an Event Broker Connection	PPG-178
Other Event Broker Activities	PPG-178
Sending Messages Between Tasks and the Event Broker	PPG-178
Message Flavors	PPG-178
Message Types	PPG-181
Flavor-Specific Message Requirements	PPG-182
The Process of Event Monitoring.....	PPG-182
Connecting a Task to the Event Broker	PPG-183
Creating a Message Port	PPG-183

Creating a Configuration Message	PPG-183
Creating a Configuration Block	PPG-183
Sending the Configuration Message	PPG-189
Receiving the Configuration Reply Message	PPG-189
Monitoring Events Through the Event Broker	PPG-190
Waiting for Event Messages on the Reply Port	PPG-190
Reading an Event Message Data Block	PPG-191
Reading Event Data	PPG-194
High-Performance Event Broker Use	PPG-199
The Pod Table	PPG-199
Gaining Access to the Pod Table	PPG-200
Relinquishing Access to the Pod Table	PPG-200
Structure of the Pod Table	PPG-200
Empty Generic Class Substructures	PPG-202
Non-Empty Generic Class Substructures	PPG-202
Use and Abuse of the Pod Table	PPG-204
Synchronization Between the Pod Table and Event Messages	PPG-206
Reconfiguring or Disconnecting a Task	PPG-206
Reconfiguring the Event Broker Connection	PPG-206
Disconnecting From the Event Broker	PPG-207
Other Event Broker Activities	PPG-207
Getting Lists of Listeners and Connected Pods	PPG-207
Working With Input Focus	PPG-210
Commanding a Pod	PPG-211
Event Broker Convenience Calls	PPG-214
Connecting to the Event Broker	PPG-214
Monitoring a Control Pad or a Mouse	PPG-215
Disconnecting From the Event Broker	PPG-216

14

The International Folio

What Is Internationalization?	PPG-217
The Locale Data Structure	PPG-218
NumericSpec Structure	PPG-220
DateSpec Arrays	PPG-223
Using the Locale Structure	PPG-223
Determining the Current Language and Country	PPG-223
Working With International Character Strings	PPG-224
Formatting Numbers or Currency	PPG-226

Formatting Dates	PPG-226
------------------------	---------

15

The Compression Folio

Introduction	PPG-227
How the Compression Folio Works.....	PPG-228
Compressing Data	PPG-228
Decompressing Data	PPG-228
How to Use the Compression Folio.....	PPG-228
The Callback Function.....	PPG-230
Controlling Memory Allocations	PPG-230
Convenience Calls.....	PPG-231
Example: Using Compression	PPG-231

16

The IFF Folio

Overview of the IFF File Format and Folio	PPG-237
The EA IFF 85 Standard	PPG-237
The IFF File Format	PPG-238
Goals of the IFF Format	PPG-239
The Purpose and Capabilities of the IFF Folio	PPG-240
How IFF Formats and Folio are Used in 3DO M2 System	PPG-243
Description of the IFF File Format	PPG-243
Standard Chunk Format	PPG-243
3DO Extension to Standard Chunk Format	PPG-245
Container Chunks	PPG-246
Using the IFF Folio - a Tutorial	PPG-250
Reading and Processing a FORM	PPG-250
Writing out a FORM	PPG-254
Description of IFF Folio Functions.....	PPG-255
Setting Up, Starting, and Terminating a Parse Operation	PPG-256
Defining, Saving, and Finding Collection Chunks	PPG-257
Defining, Saving, and Finding Property Chunks	PPG-259
Defining Stop Chunks	PPG-262
Defining Entry and Exit Handlers	PPG-263
Using ContextInfo Structures	PPG-264
Finding Context Nodes	PPG-268
Pushing and Popping Context Nodes	PPG-268

Reading and Writing Chunks	PPG-269
Positioning the Cursor in a Stream	PPG-270

17

The Icon Folio

Introduction - What the Icon Folio Does	PPG-271
About Icons	PPG-271
Functions Provided by the Icon Folio	PPG-272
Loading an Icon into Memory	PPG-272
Unloading an Icon from Memory	PPG-274
Saving an Icon	PPG-274

18

The SaveGame Folio

Introduction - What the SaveGame Folio Does	PPG-275
What Kind of Data is Saved - Determined by Application	PPG-275
Functions Provided by the SaveGame Folio	PPG-276
Saving Game Data	PPG-276
Loading Saved Game Data into Memory	PPG-278

19

The Requestor Folio

Introduction - What the Requestor Folio Does	PPG-281
Purpose	PPG-281
How to Use the Requestor Folio - Overview	PPG-282
Creating a Requestor Object	PPG-283
Displaying the Storage Requestor Interface to the User	PPG-285
Querying a Storage Requestor Object	PPG-285
Modifying a Storage Requestor Object	PPG-286
Deleting a Storage Requestor Object	PPG-286

20

Dynamic-Link Libraries

About DLLs	PPG-287
Creating and Using 3DO M2 DLLs	PPG-288
Modules	PPG-288
Linking Multi-Module Applications	PPG-289
Definition Files	PPG-290
How Definition Files Work	PPG-290
Example of a Definition File	PPG-291

Building a Multi-Module Application	PPG-291
Files Used in a Multi-Module Application	PPG-292
Building an Exporting Module	PPG-294
Building and Executing an Importing Module	PPG-295
Linking an Application with a DLL	PPG-295
Functions for Handling Modules	PPG-296
Using Items as Arguments and Return Values	PPG-296
Opening, Loading, and Executing Modules	PPG-296
Importing and Exporting Modules	PPG-297
Removing Symbols from the List of Available Symbols	PPG-299
Looking Up the Address of a Symbol	PPG-299
Example: Creating and Using a DLL.....	PPG-300
Building a DLL Step by Step	PPG-300
Importing a DLL Step by Step	PPG-301
Executing the IMPORTER Program	PPG-301
Output of the IMPORTER Program	PPG-301

21

The Debug Console Link Library

About the Debug Console Link Library	PPG-311
Preparing For Console Output	PPG-312
Performing Console Output	PPG-312
Concluding Console Output	PPG-313

22

The Script Folio

About the Script Folio	PPG-315
Executing Commands	PPG-316
Preserving State Across Multiple Executions	PPG-316

23

Using Lumberjack, the Event Logger

Overview	PPG-319
Collecting Information With Lumberjack	PPG-320
Creating Lumberjack	PPG-320
Starting and Stopping Event Logging	PPG-320
Logging Custom Events	PPG-320
Deleting Lumberjack	PPG-321
Getting Information From Lumberjack.....	PPG-321

Getting and Releasing Buffers	PPG-321
Parsing Lumberjack Buffers	PPG-322
Example: Using Lumberjack	PPG-323

Index.....	PPG-325
-------------------	----------------

List of Figures

- Figure 1-1: Kernel page allocation. PPG-6
- Figure 1-2: Contiguous allocation of 3KB memory block. PPG-7
- Figure 1-3: Non-contiguous allocation of 3KB memory block. PPG-8
- Figure 4-1: Anchored list. PPG-38
- Figure 13-1: Pod table structure fixed-size and fixed-format section. PPG-201
- Figure 13-2: Pod substructure. PPG-202
- Figure 13-3: Generic class substructure. PPG-203
- Figure 16-1: Layout of a sample AIFC FORM PPG-239
- Figure 16-2: A FORM with 2 embedded FORMs containing local chunks PPG-241
- Figure 16-3: Context stack when local chunk F2LC is being read PPG-242
- Figure 16-4: Context stack when local chunk F3LC is being read PPG-242
- Figure 16-5: Layout of a sample TXTR FORM PPG-245
- Figure 16-6: 3DO extended format PPG-246
- Figure 19-1: The Storage Manager (Requestor) Interface PPG-282
- Figure 20-1: Building an exporting module PPG-293
- Figure 20-2: Building an importing module PPG-294

List of Examples

Example 2-1: Starting a task. PPG-25
Example 2-2: Using threads (signals.c). PPG-28
Example 3-1: Convention often used to document tag args. PPG-33
Example 3-2: Specifying tag arguments in a tag array PPG-34
Example 3-3: Specifying tag arguments in a VarArg list PPG-35
Example 4-1: The Note Tracker structure in the music library. PPG-39
Example 4-2: Traversing a list front to back. PPG-43
Example 4-3: Traversing a list back to front. PPG-44
Example 4-4: The Node data structure. PPG-47
Example 4-5: The NamelessNode structure. PPG-47
Example 4-6: The MinNode structure. PPG-48
Example 4-7: The List data structure. PPG-48
Example 4-8: The ListAnchor union. PPG-49
Example 4-9: The Link data structure. PPG-49
Example 5-1: Allocating memory. PPG-61
Example 6-1: ItemNode structure PPG-68
Example 6-2: Defining values for an item's associated tag argument. PPG-70
Example 6-3: Tag arguments array. PPG-71
Example 6-4: A standard function call with TagArg structures. PPG-72
Example 6-5: Using VarArgs to provide tag arguments. PPG-72
Example 8-1: Allocating a signal bit. PPG-85
Example 8-2: Samples using the message passing routines (msgpassing.c). PPG-96
Example 9-1: IOInfo, IOBuf structures PPG-108
Example 9-2: Looking at fields of the message structure PPG-112
Example 9-3: IOReq structure PPG-112
Example 10-1: TimeVal structure PPG-117

- Example 10-2: Initializing `IOInfo` for `TIMERCMD_DELAYUNTIL_USEC` PPG-119
- Example 10-3: Initializing `IOInfo` for `TIMERCMD_DELAYUNTIL_VBL` PPG-121
- Example 10-4: Initializing `IOInfo` for `TIMERCMD_DELAY_VBL` PPG-122
- Example 10-5: Initializing `IOInfo` for `FILECMD_FSSTAT` PPG-125
- Example 10-6: Initializing `IOInfo` for `CMD_STATUS` PPG-126
- Example 10-7: Initializing `IOInfo` for `FILECMD_ALLOCBLOCKS` PPG-127
- Example 10-8: Initializing `IOInfo` for `CMD_BLOCKWRITE` PPG-127
- Example 10-9: Initializing `IOInfo` for `FILECMD_SETEOF` PPG-128
- Example 10-10: Initializing `IOInfo` for `CMD_BLOCKREAD` PPG-128
- Example 10-11: Initializing `IOInfo` for `FILECMD_SETTYPE` PPG-129
- Example 10-12: Initializing `IOInfo` for `FILECMD_SETVERSION` PPG-129
- Example 10-13: Initializing `IOInfo` for `FILECMD_SETBLOCKSIZE` PPG-130
- Example 10-14: Initializing `IOInfo` for `FILECMD_READDIR` PPG-130
- Example 10-15: Initializing `IOInfo` for `FILECMD_READENTRY` PPG-131
- Example 10-16: Initializing `IOInfo` for `CMD_STREAMWRITE` PPG-132
- Example 10-17: Initializing `IOInfo` for `CMD_STREAMREAD` PPG-132
- Example 10-18: Initializing `IOInfo` for `SER_CMD_SETCONFIG` PPG-133
- Example 10-19: Initializing `IOInfo` for `SER_CMD_GETCONFIG` PPG-134
- Example 10-20: Initializing `IOInfo` for `SER_CMD_STATUS` PPG-136
- Example 11-1: Displaying the contents of a 3DO file (`type.c`) PPG-155
- Example 11-2: Scanning the file system (`Walker.c`). PPG-159
- Example 11-3: Listing a directory (`ls.c`) PPG-162
- Example 11-4: Appending pathnames PPG-170
- Example 12-1: `GregorianCalendar` structure. PPG-172
- Example 12-2: `TimeVal` structure. PPG-173
- Example 13-1: `ConfigurationRequest` structure. PPG-184
- Example 13-2: `FindMsgPort()`. PPG-189
- Example 13-3: `EventFrame` structure PPG-192
- Example 13-4: `ControlPadEventData` structure PPG-194
- Example 13-5: `LightGunEventData` structure PPG-197
- Example 13-6: `StickEventData` structure PPG-198
- Example 13-7: `FilesystemEventData` structure PPG-199
- Example 13-8: `PodDescriptionList` structure PPG-208
- Example 13-9: `PodDescription` structure PPG-208
- Example 13-10: `ListenerList` structure PPG-210
- Example 13-11: `SetFocus` structure PPG-211
- Example 13-12: `PodData` structure PPG-212
- Example 14-1: `Locale` structure PPG-219

Example 14-2: NumericSpec structure PPG-221
Example 15-1: Compressing and decompressing data (compression.c). PPG-231
Example 16-1: Format of IFF chunk PPG-244
Example 16-2: Outline of a CAT chunk containing FORM chunks PPG-247
Example 16-3: Outline of a LIST chunk containing PROP and FORM chunks PPG-248
Example 16-4: Outline of the VID1 FORM used for the tutorial PPG-251
Example 16-5: CollectionChunk structure PPG-257
Example 16-6: IFFTypeID structure PPG-258
Example 16-7: Outline of a LIST containing another LIST, which contains FORMs PPG-260
Example 16-8: Top, top property, and bottom nodes in a sample context stack PPG-264
Example 16-9: ContextInfo structure PPG-265
Example 17-1: In-memory representation of an icon PPG-274
Example 18-1: The SGDATA structure PPG-278
Example 19-1: Option flags for use with the STORREQ_TAG_OPTIONS tag PPG-284
Example 20-1: Format of a Definition File PPG-291
Example 20-2: Flags used to Link Modules PPG-292
Example 20-3: The EXPORTER.C File PPG-302
Example 20-4: EXPORTER.H: The EXPORTER program's header file PPG-303
Example 20-5: The IMPORTER.C File PPG-304
Example 20-6: The EXPORTER.X File PPG-306
Example 20-7: The EXPORTER.MAKE File (Generated by MPW) PPG-306
Example 20-8: The IMPORTER.MAKE File (Generated by MPW) PPG-308
Example 23-1: Using Lumberjack PPG-323

Preface

This Preface introduces the chapters covered in this book, lists related documentation, and explains typographical conventions used in the text.

This Preface contains the following topics:

Topic	Page Number
About This Book	xxiii
Programmer's Guide vs. Programmer's Reference	xxiv
How This Book is Organized	xxiv
Related Documentation	xxv
About the Code Examples	xxvi
About Bit Ordering	xxvii
Typographical Conventions	xxvii

About This Book

The *3DO Operating System Programmer's Guide* is a guide to the features of Portfolio, the 3DO operating system. It includes overviews, programming tutorials, and sample code.

This book contains descriptions of the system level components that make up Portfolio: kernel, I/O, tasks, filesystem, events, and so on. It is written for title developers who create programs that run on a 3DO M 2 system.

To use this document, the developer should have a working knowledge of the C programming language.

Programmer's Guide vs. Programmer's Reference

This book, the *3DO Operating System Programmer's Guide*, describes concepts and methodologies. It does not contain detailed descriptions of all the Portfolio function calls.

For detailed descriptions of the Portfolio function calls, see the *3DO M2 Portfolio Programmer's Reference*, which should be used in conjunction with this book.

How This Book is Organized

This book contains the following chapters:

- ◆ **Chapter 1, "Understanding the Kernel"** — Gives an overview of the role of the kernel in the operation of Portfolio.
- ◆ **Chapter 2, "Tasks and Threads"** — Gives the background and necessary programming details to create and run tasks and threads. Note that tasks and threads, although similar, operate differently in certain key respects.
- ◆ **Chapter 3, "Using Tag Arguments"** — Provides background on tag commands and tag arguments and explains how to use them.
- ◆ **Chapter 4, "Managing Linked Lists"** — Provides background on Portfolio linked lists and explains how to manage them.
- ◆ **Chapter 5, "Managing Memory"** — Explains how to allocate memory, how to share it among tasks, how to get information about it, and how to free it.
- ◆ **Chapter 6, "Managing Items"** — Explains what items are and how to use them.
- ◆ **Chapter 7, "Semaphores"** — Explains techniques for using semaphores to share system resources.
- ◆ **Chapter 8, "Communicating Among Tasks"** — Provides background and necessary programming details for doing intertask communication.
- ◆ **Chapter 9, "The Portfolio I/O Model"** — Provides an overview of 3DO hardware devices and explains how to handle input/output operations using the standard Portfolio I/O calls.
- ◆ **Chapter 10, "Portfolio Devices"** — Lists the device drivers and their associated commands and options.
- ◆ **Chapter 11, "The File System, File Folio, and FSUtils Folio"** — Describes the 3DO file system. It shows how to use calls from both the File folio and the FSUtils (File System Utilities) folio.
- ◆ **Chapter 12, "The Batt Folio and Date Folio"** — Describes the battery-backed clock, which end-users can use to display local time, and the calls that support it.

- ◆ **Chapter 13, "The Event Broker"** — Shows how a task uses the event broker to work with interactive devices.
- ◆ **Chapter 14, "The International Folio"** — Provides information on how an application can use the International folio to determine the current language and country, and how to display dates, currency, and numeric values in a manner consistent with the current language and country codes.
- ◆ **Chapter 15, "The Compression Folio"** — Describes the Compression folio, which provides general-purpose compression and decompression services.
- ◆ **Chapter 16, "The IFF Folio"** — Describes the IFF file formatting standard and the folio used by 3DO to simplify reading and writing IFF files. The IFF standard facilitates storing, retrieving, and parsing complex information.
- ◆ **Chapter 17, "The Icon Folio"** — Describes the Icon folio, which facilitates storing icons in files and loading icons into memory for rendering.
- ◆ **Chapter 18, "The SaveGame Folio"** — Describes the SaveGame folio, which facilitates storing game data into files and loading it into memory.
- ◆ **Chapter 19, "The Requestor Folio"** — Describes the Requestor folio, which provides a ready-made graphical user interface that you can use to let your end-users interact with 3DO file systems.
- ◆ **Chapter 20, "Dynamic-Link Libraries"** — Describes how to set up and use libraries of executable modules that are loaded dynamically at run time rather than statically at link time.
- ◆ **Chapter 21, "The Debug Console Link Library"** — Describes a set of functions that lets you send debugging output to a View on the 3DO display itself instead of to the 3DO debugger. This lets you debug a program with minimum performance degradation.
- ◆ **Chapter 22, "The Script Folio"** — Describes the Script folio functions, which let you execute Portfolio shell commands from within your program.
- ◆ **Chapter 23, "Using Lumberjack, the Event Logger"** — Describes Lumberjack, which is a collection of kernel functions that lets you log events for debugging.

Related Documentation

The following additional manuals are useful to developers who are programming in the 3DO M2 environment.

- ◆ **3DO M2 Release 2.0 Global Table of Contents** — Contains a global, composite table of contents for all the 3DO M2 Release 2.0 manuals.
- ◆ **3DO M2 Release 2.0 Global Index** — Contains a global, composite index for all the 3DO M2 Release 2.0 manuals.

- ◆ *Getting Started with 3DO M2 Release 2.0* — Describes initial setup of the M2 system for developers.
- ◆ *3DO M2 Portfolio Programmer's Reference* — Contains a detailed description of the calls that make up the Portfolio system.
- ◆ *3DO M2 Graphics Programmer's Guide* — Describes the M2 graphics system, a layered set of APIs for managing the M2 rendering and display hardware. This book contains chapters on all the components of the M2 graphics system: The Graphics Framework, the Graphics Pipeline, the Graphics Folio, the 2D Graphics Framework, and M2 SDF (scene description format) files, which can be used to describe scenes used in M2 applications. The *3DO M2 Graphics Programmer's Guide* provides code examples that show how the various components in the M2 graphics system work, and shows how you can use the various M2 graphics components in developing your own titles.
- ◆ *3DO M2 Graphics Programmer's Reference* — Contains descriptions and syntax listings of all function calls used to access the various components described in the *3DO M2 Graphics Programmer's Guide*.
- ◆ *3DO M2 Audio and Music Programmer's Guide* — Describes the music and audio folios. It includes overviews, programming tutorials, sample code, and procedure call definitions.
- ◆ *3DO M2 Audio Programmer's Reference* — Contains a detailed description of the calls that make up Portfolio music and audio.
- ◆ *3DO M2 Command List Toolkit* — Describes the Command List Toolkit and it's relationship to the M2's 3D hardware rendering engine.
- ◆ *3DO M2 Video Processing Guide* — Describes how to prepare and process video materials into digital video clips for 3DO M2 title development, discusses hardware options, and suggests a number of software solutions.
- ◆ *3DO M2 Debugger Programmer's Guide* — A guide to using the 3DO M2 Debugger. This book provides a tutorial on using the Debugger as well as descriptions of the graphical user interface.
- ◆ *3DO M2 DataStreamer Programmer's Guide* — A guide to the 3DO M2 DataStreamer architecture, streaming tools, and Weaver tool.
- ◆ *3DO M2 DataStreamer Programmer's Reference* — A command reference for the 3DO M2 DataStreamer, streaming tools, and Weaver tool.
- ◆ *DIAB Data Compiler* — Contains the *Language User's Manual*, the *PowerPC Target User's Manual*, the *PowerPC Assembler User's Manual*, and the *Utilities User's Manual*.

About the Code Examples

The examples used in this book are provided in electronic format in the Examples folder of the release CD.

About Bit Ordering

This book uses traditional C-style bit ordering, in which Bit 0 refers to the least significant bit. This convention is the opposite of that used in the PowerPC documentation, where Bit 0 is the most significant bit.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>Err_OpenCompressionFolio(void);</code>
procedure name	<code>CreateSmallMsg()</code>
new term or emphasis	That added weight is called a <i>cornerweight</i> .
tag argument	<code>SAVEGAMEDATA_TAG_FILENAME (const char *)</code> (See "How Tags are Documented" on page 33.)

Understanding the Kernel

This chapter describes the role of the kernel in the 3DO operating system.

This chapter contains the following topics:

Topic	Page Number
Description of the Kernel	1
Multitasking	2
Memory Management	5
Working with Folios	10
Managing Items	10
Semaphores	12
Intertask Communication	13
Portfolio Error Codes	16

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

Description of the Kernel

The kernel is a collection of function calls that is the heart of the Portfolio operating system. The kernel controls the 3DO hardware and any software running on the system. It manages resources and provides communication between running tasks and hardware devices. In particular, the kernel handles these responsibilities:

- ◆ **Multitasking.** The kernel handles the execution of all programs (tasks) running on the system and provides multitasking so that many tasks can run simultaneously.
- ◆ **Memory Management.** The kernel allocates memory to all running tasks and makes sure that one task doesn't indiscriminately write to the memory allocated to another task. The kernel also consolidates free memory, manages the system's own memory, and performs other memory management duties.
- ◆ **Folio management.** Folios are the mechanism used by the kernel to bundle related functionality. Portfolio is composed of many folios, which together provide the system API. The kernel manages the creation, disposal, loading, and unloading of folios.
- ◆ **Intertask Communication.** The kernel enables multiple tasks running simultaneously to communicate with one another. The two primary methods of intertask communication are by sending high-performance boolean signals, or by sending messages.
- ◆ **Resource sharing.** The kernel provides a semaphore system for critical resources that tasks share so that only one task at a time works with those resources.
- ◆ **Portfolio Error Codes.** The kernel provides a consistent way to handle error returns throughout the system. It helps convert error codes into descriptive strings to make application development easier.
- ◆ **I/O.** The kernel provides synchronous and asynchronous communication with I/O devices, and includes device and driver definitions for those devices.

Multitasking

One of the most important jobs of the kernel is managing the tasks that run on a 3DO system. Because Portfolio supports pre-emptive multitasking, the kernel must provide conventions for deciding which tasks get CPU time, and for saving a task's state when execution switches from one task to another.

Multitasking

The kernel supports pre-emptive context-switching for multitasking. The kernel normally devotes one time quantum (normally 15 milliseconds) of CPU time to a task and then switches execution to another task, where it devotes another time quantum before switching to yet another task. To make this switch without destroying the current state of the executing task, the kernel saves the context of the current task in the task's TCB (task control block, a data structure that contains the parameters of each task), and reads the context of the next task from its control block before executing that task. (A task's context is its state, which includes its allocated address space and its register set.)

At any point during a time quantum, whenever the need arises, the kernel can preempt the current task and immediately switch execution to a more important task. To determine how and when it should switch from one task to another, the kernel reads task states and priorities.

Task States

A task can be in one of three states:

- ◆ **Running:** The task is currently executing in the current time quantum.
- ◆ **Ready to run:** The task is stored in the *ready queue*, awaiting execution by the kernel in a future time quantum.
- ◆ **Waiting:** The task is asleep waiting for an external event (such as a vertical blank or an I/O request) to occur. Once the task is notified of the event's occurrence, the task moves to the ready queue for execution.

The kernel executes tasks in the ready queue only, switching from task to task as required. It does not execute waiting tasks, so waiting tasks don't require any CPU cycles—a courtesy to the other tasks running on the system.

To determine how the ready-to-run tasks are executed, the kernel considers each task's priority.

Task Priorities

Each task is associated with a priority. The priority of a task is a value ranging from 10 to 199, with 10 being the lowest priority and 199 being the highest. The priority of a task can be changed at any time to give the task a higher or lower priority than others in the ready queue—or to give the task an equal priority with other tasks.

Priority determines the order in which tasks in the ready queue are executed. The kernel executes only the highest-priority task (or tasks) in the ready queue and doesn't devote any CPU time to lower-priority tasks until all higher-priority tasks in the ready queue are completed. When higher-priority tasks are scheduled for execution, a lower-priority task cannot execute until one of the following conditions is met:

- ◆ All higher-priority tasks have finished execution and have exited the system.
- ◆ A higher-priority task cannot execute immediately because it has to wait for an event.
- ◆ A higher-priority task is demoted to a lower priority.

If several tasks all share the same highest priority, the kernel cycles among those tasks using a process known as round-robin scheduling. The kernel's round-robin strategy gives one time quantum to each task competing for processing time. Whenever a new task with a higher priority than the one running enters the ready

queue, or whenever an existing task is given a higher priority than the one running, the kernel preempts the running task. It immediately switches execution to the higher-priority task, even if the switch occurs in the middle of a time quantum. The higher-priority task then starts at the beginning of its own time quantum.

Note that round-robin scheduling takes place only when tasks of equal priority have the highest priority in the ready queue. If only one task has the highest priority, only that task runs, and all others languish until it finishes or is kicked out of the CPU limelight by another task with a higher priority. Note also that if a task executes a `Yield()` call, it forgoes the rest of its quantum, and yields the CPU immediately to other tasks of equal priority.

Waiting Tasks

To wait for an event, a task must execute a wait function call (such as `WaitSignal()`, `WaitIO()`, or `WaitPort()`) to define what it's waiting for. It then becomes a waiting task and receives no CPU time. When its wait conditions are satisfied, a task moves to the ready queue, where it can compete for CPU time.

Waiting tasks are an important feature for keeping running tasks working at top speed without wasting CPU cycles on tasks waiting for external events. For the system to run at top efficiency, each task must *not* use a loop that constantly checks for an event. Repeatedly checking for an event, known as "busy-waiting," is not recommended because it consumes unnecessary CPU cycles.

Task Termination

As each task runs, it accumulates its own memory and its own set of resources. The kernel keeps track of the memory and resources allocated to each task and when that task quits or dies, the kernel automatically closes those resources and returns the memory to the free memory pool. This means that the task doesn't have to close resources and free memory on its own before it quits, a convenient feature. Good programming practice, however, is to close and release resources and free memory within a task whenever they are no longer in use. If a thread exits, it must free its memory so that the parent task can allocate it for its own use.

Parent Tasks, Child Tasks, and Threads

Any task can launch another task. The launched task then becomes the child of the launching task and the child task is a resource of the parent task. This means that when the parent task quits, all its children quit too.

To sever the parent-child relationship between two tasks so that the child task doesn't quit with the parent task, the parent can use the `SetItemOwner()` function call to transfer ownership of the child task to the child task itself. Then, when the parent task quits, the child task continues to run.

A parent task spawns child tasks to take care of real-time processing and other operations. A child task has one big disadvantage: it doesn't share memory with the parent. Because it must allocate its own memory in pages, it can waste memory if its memory requirements are small. And because parent and child don't share memory, they can't share values stored in shared data structures. To overcome these disadvantages, a parent task can spawn a *thread*.

A thread is a child task that shares the parent task's memory. The owner of the thread can transfer ownership of a thread to any thread in the parent's task family except to the thread itself. A task family is a task and all of its threads.

Memory Management

A multitasking system must manage memory carefully so that one task doesn't write to another's memory. To prevent this kind of disaster, the kernel allocates exclusive memory to each task and restricts each task to writing to its own memory unless given specific permission to do otherwise.

Memory Size

Memory in the 3DO system is divided into *pages*. Currently, each page holds 4KB of RAM. Although a memory page currently holds 4KB of RAM, the 3DO system's memory page size is always subject to change, depending on the current hardware design. Do not rely on any specific memory-page size when you design your titles. If you need to know the current page size, call the `GetPageSize()` function.

Allocating Memory

Whenever a task starts from a file, the file includes the memory requirements of the task—that is, how much code space and how much data space the task requires to run. The memory requirements of a task are set when the task's source code is compiled; they are determined by the sizes of dimensioned arrays and other factors. The kernel automatically allocates the appropriate amount of memory to each task before the task loads and runs. If a task requires extra memory once it is running, it must request the memory by calling kernel memory-allocation functions such as `AllocMem()` or `malloc()`.

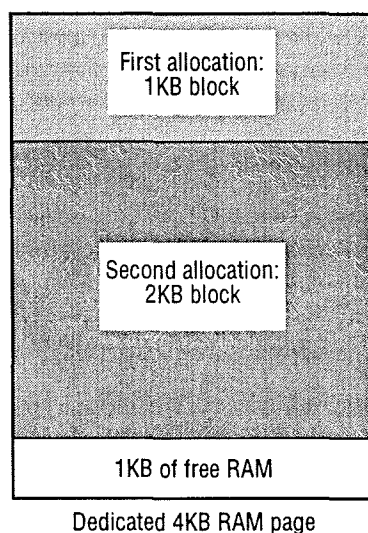


Figure 1-1 *Kernel page allocation.*

The kernel dedicates memory to a task a page at a time. Then it allocates memory to the task from within the pages it has dedicated to the task. For example, suppose a task requires 1 KB of memory. When the task starts, the kernel reserves a single 4 KB page for the task by putting a *fence* around it. (For more information about fences, see “Memory Fences” on page 8.) Because the task being started requires 1KB of memory, the kernel allocates the first 1 KB of the reserved 4KB page to the task when the task starts, as shown in Figure 1-1.

Although there is a fence around the full 4KB page reserved for the task, the remaining 3 KB of RAM on the page dedicated to the task remain unallocated.

Now suppose the task requests another 2 KB of memory. The kernel allocates the next 2 KB of the dedicated page to the task. That allocation still leaves 1 KB of RAM in the page free for future allocation, as shown in Figure 1-1 on Page 6.

When a task requests a block of memory that is larger than any contiguous stretch of free RAM left in the pages reserved for the task, the kernel dedicates one or more new pages of RAM to the task – if it can find a set of contiguous pages large enough to handle the request. The kernel joins these new pages with the memory already in the task’s free list.

The kernel then provides the task with the memory it has requested by allocating RAM from the new pages dedicated to the task. In this way, the kernel dedicates as many pages of RAM to the task as the task has requested, if possible. If the kernel cannot find enough contiguous pages to allocate all the requested memory, it notifies the task that the memory cannot be allocated.

To see how the extension of a memory allocation works, consider the example discussed earlier in this section. Recall that a certain task required only 1 KB of RAM at startup, but began with a 4KB page because memory in the 3DO system is divided into 4KB pages. The task then requested and received another 2 KB of RAM from the same page.

Now suppose that the same task requests still another 3 KB of RAM. As Figure 1-1 shows, there is now only 1 KB of free on the first page dedicated to the task. So the kernel dedicates another page of RAM to the task. The kernel then fulfills the task's request for an additional 3 KB of RAM by allocating 3 KB of RAM to the task from the newly dedicated page.

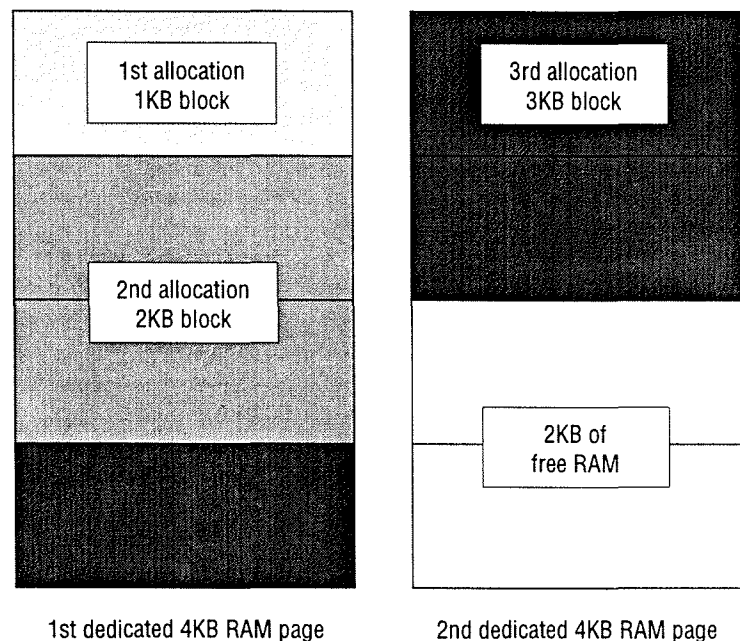


Figure 1-2 *Contiguous allocation of 3KB memory block.*

This newly reserved page may or may not be contiguous. If the new page is contiguous, the allocation will be done as shown in Figure 1-2. If the new page is not contiguous, the allocation will be done as shown in Figure 1-3. In the latter case, the kernel leaves a 1KB block of RAM unallocated on the first page and allocates all requested RAM from the newly dedicated page.

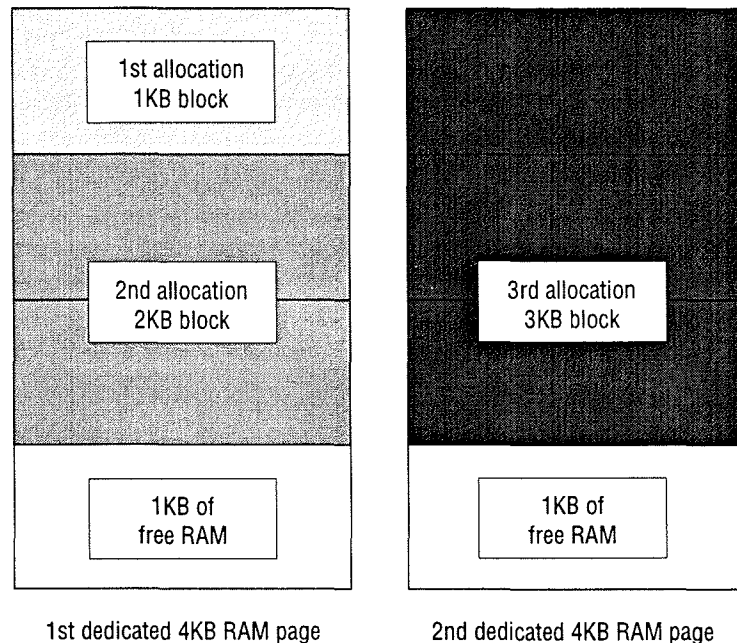


Figure 1-3 *Non-contiguous allocation of 3KB memory block.*

Any allocated block of memory must be on contiguous pages. If the system cannot find enough contiguous pages to accodate the allocation request, the request fails.

Memory Fences

When the kernel dedicates one or more pages of memory to a task, it sets up a *memory fence* around the dedicated memory. The task can then write to any address within memory that is fenced in, but normally it can't write outside its dedicated memory. There is only one exception: A user task can write to memory reserved for use by a second user task provided the second user task specifically grants permission. The first task can write there *only* with permission from the second task; if it doesn't have permission, it can't write there.

Also, a user task can never write directly to system memory (memory allocated to system tasks) because system tasks never grant write permission to user tasks.

When a task writes (or tries to write) to RAM addresses that have not been allocated to it, the attempt can have one of two results:

- ◆ If the task tries to write outside its fence, the kernel aborts the task.
- ◆ If the task writes inside its fence, the task isn't aborted, but it can write over its own data, causing unforeseen problems.

Although fences restrict tasks from writing outside of their allocated memory, they don't restrict tasks from reading memory elsewhere in the system. A user task can read memory allocated to other user tasks as well as to system tasks.

Returning Memory

The kernel keeps track of free RAM in two ways: it keeps a list of all RAM pages allocated to tasks (and so it knows which pages are free) and it lists free blocks of RAM within the dedicated pages. Whenever a task is finished with a block of allocated RAM, the kernel can free the block for further allocation by using a function call that specifies the beginning address and size of the block. (The Kernel folio includes a number of memory-freeing calls, such as `FreeMem()` and `free()`.) If all the allocated blocks within a dedicated page of RAM are freed, the kernel knows that the page is free but keeps the page dedicated to the task in case there are future allocation calls.

When a task wishes to release free RAM pages back to the kernel so the kernel can dedicate them to other tasks, the task issues a `ScavengeMem()` call. This call causes the kernel to reclaim free pages and to list them once again in the system free-page pool.

Whenever a task quits or dies, all of its memory returns to the free RAM pool—*unless* it's a thread. When a thread dies, its memory remains the property of the parent task, because a thread shares the memory of its parent task.

Sharing Memory

The kernel gives every page of memory dedicated to a task a status that tells which task owns the page and whether other tasks can write to that page or not. The status is set so that only the owning task can write to the page. If the owning task wants to share write privileges with another task (or with several other tasks), it can change the status with the `ControlMem()` call, which can take three actions:

- ◆ Specify another task and give it write privileges to the page.
- ◆ Specify another task and retract its write privileges to the page.
- ◆ Specify another task and give it ownership of the page.

As long as a task owns a page, it can change the page status to any state it specifies—it can turn write privileges on and off for other tasks or even for itself. However, once a task transfers ownership of a page to another task, the original task can no longer set that page's status, which is under the sole control of the new owner task. If the original task tries to write to the page, it aborts. Any I/O operation using that page as a write buffer also aborts.

Working with Folios

Portfolio is organized into *folios*. Each folio is a collection of function calls dealing with a different aspect of 3DO operation.

When a task needs a folio, the folio is fetched from disk automatically. When a task tries to open a folio, the kernel first looks for the folio in memory. If the kernel does not find the folio in memory, the kernel loads it from disk. Once a folio has been loaded from disk so a task can use it, the folio can be used by other tasks in the system without having to be reloaded.

When a folio is no longer in use by any task, the kernel tries to keep the folio loaded in memory as long as possible in case another task wants to use it. If memory runs low in the system however, the kernel will remove unused folios to free up memory.

Managing Items

In a multitasking system, tasks often share resources such as RAM, I/O devices, and data structures. If the system doesn't manage those resources carefully, a task that depends on a shared resource can be brutally disappointed. For example, one task creates a data structure for a second task to read and act upon. If the first task dies and its data structure is no longer maintained or ceases to exist, the second task can find itself reading erroneous data that can crash it. In another example, a task creates a data structure and then uses an invalid pointer to that structure (typically a NULL or uninitialized pointer). Because the pointer doesn't point where it should, the task can read totally erroneous data or, even worse, try to write to the data structure and crash itself.

Items

To ensure the integrity of shared resources, the kernel provides *items*. An item is a system-maintained handle for a shared resource. The handle contains a globally unique item number and a pointer to the resource. When a task needs access to a shared resource, the task simply asks for the resource by item number instead of using a pointer to point directly at the resource. The kernel checks its list of items and, if it finds the requested item, performs any requested actions on the item. If the kernel finds that the item no longer exists, it informs the requesting task that access failed, and the task can go on without crashing itself or the system.

An item can refer to any one of many system components: a data structure used to create graphics, an I/O device, a folio, or even a task itself. In fact, almost all system-standard structures must be created as items. You'll find one of the most commonly used kernel calls is `CreateItem()`, which, appropriately enough, creates items. You use it to start a task, to lay out a graphics environment, to create messages to send between tasks, and to handle many other Portfolio functions.

Creating an Item

To create an item, a task typically creates a list of parameters (tag arguments) for the item. The parameters can include a name for the item, its dimensions and contents, and other important information. The task then uses `CreateItem()` to ask the kernel to create the item. The task supplies an item-type number that specifies the type of item to create, and a pointer to the list of parameters for the item. The kernel has a predefined set of item types (which you can find in the *3DO M2 Portfolio Programmer's Reference*). Each item requires different parameters to define it.

The kernel receives the item parameters, and—if they're correct for the specified item type—creates the item. The item contains a globally unique ID number, the item type, a pointer to the resource handled by the item, and other parameters passed to it. The kernel returns the item number to the task that created the item. The kernel records the fact that the item belongs to that task.

Portfolio provides a large number of convenience calls that let you easily create items; for example, `CreateMsgPort()` and `CreateThread()`. It is generally simpler to use these higher-level routines instead of calling `CreateItem()` directly.

Opening Items

Many system items are predefined and stored on the system boot disk. They don't need to be created from scratch, only opened. To open a system item, a task uses the `FindAndOpenItem()` call to specify the type of the item to open, provides an array of the arguments required to open the item, and passes that information along to the kernel to open the item. The kernel finds the item on the disk, brings necessary data structures into system memory, assigns the item a number, and returns the item number to the task that asked for the item to be opened. When the task is finished with the opened item, it uses the `CloseItem()` call to close the item.

Using Items

Once an item is opened or created, tasks can use it by specifying its item number. If a task doesn't have the number for an item it wants to use, or knows the item number but doesn't know what type of item it is, the kernel provides item-handling function calls such as `FindItem()`, `FindNamedItem()`, and `LookupItem()`. These calls help find items by name, number, or other criteria, and then return item type, item number, or other information about the item.

The kernel provides the `SetItemPri()` call to change the priority of an item within a list of items. It also provides the `SetItemOwner()` call to change ownership of an item.

Deleting Items

Whenever a task finishes using an item that it created, it can delete the item with `DeleteItem()`, which removes the item and frees any resources committed to it, such as RAM devoted to the item's data structure. The deleted item's number is not recycled when new items are created, so the kernel can inform tasks trying to use that item number that the item no longer exists.

There is one important rule about item deletion: a task can't delete any item it doesn't own. This means that if a task creates an item and then passes ownership to another task, the first task can't delete the item—only the new owner can delete the item.

Whenever a task or thread quits or dies, the kernel automatically deletes all items that it created, and closes any items that it opened.

Semaphores

Tasks running on a 3DO system can share data structures, storing a structure in one task's memory and allowing one or more outside tasks to read and write to that structure. Because one task using the data structure can never be sure what another task may be doing to the same structure, dangers arise. For example, suppose one task writes to a data structure at the same time that another task writes to it. The task then overwrites the first task's data, and neither task is aware of what has happened. To avoid conflicts like this, a task must be able to lock down a shared data structure while it is working on it, and then release the structure when it's done.

To provide a lock, the kernel supplies an item called a *semaphore*. A semaphore is an element of a data structure; you can think of the semaphore as an "occupied" sign for that data structure. A well-behaved task checks the semaphore of a data structure before it uses the structure. If the semaphore shows that the structure is currently unused, the task can go to work on the structure. If the semaphore shows that the structure is in use, the task must either wait until the semaphore says the structure is free or return to execution without using the structure.

To use a data structure with a semaphore, a task makes the function call `LockSemaphore()` to lock the structure's semaphore. If the semaphore is unlocked, the kernel locks it and the task can proceed with its business, using the semaphored data structure as it sees fit without interference from other polite tasks. When the task is done, it makes another function call, `UnlockSemaphore()`, to unlock the semaphore, releasing the semaphored data structure for use by other tasks.

When a task calls `LockSemaphore()`, it specifies what it will do if the semaphore is already locked: wait for the semaphore to be unlocked (putting itself to sleep); or return to execution without using the semaphore structure. If the semaphore is locked, the task acts accordingly.

The semaphore is a completely voluntary mechanism; it is what its name implies, only a flag that tells whether a data structure is in use or not. It does *not* deny write permission to tasks that want to use the data structure without checking the semaphore. If you want to share a data structure with another task, be sure that the other task is written to check for the semaphore before it goes about its business.

Intertask Communication

When one task needs to communicate with another task, it can be a simple matter of notification (“Hey! I’m finished doing what you asked me to do”) or a more involved passing of detailed information (“Here’s the table of values you asked me to calculate”). Portfolio provides mechanisms to handle both: *signals* for simple notification and *messages* for passing detailed information.

Signals

A signal is a kernel mechanism that allows one task to send a flag to another task. The kernel dedicates one 32-bit word in a task’s TCB (task control block) for receiving signals. The kernel writes to that word each time it carries a signal to the task.

The 31 least-significant bits of the 32-bit signal word each specify a different signal; the most-significant bit is used for errors. Eight of the signal bits (0 to 7) are reserved by the kernel for system signals; the other 23 bits (8 to 30) are left for custom user-task signals.

Allocating Signals

A task can’t receive signals from another user task unless it has allocated a signal bit. To do so, it uses the `AllocSignal()` call, which returns a 32-bit value containing the next unused user signal bit (the 32-bit value is called a *signal mask*). The task can give its signal mask to other tasks; they can then send a signal back to the task, using the user signal bit set in its signal mask.

All tasks can receive system signals at any time. The lower 8 signal bits are reserved for this purpose. For example, the system sends a task a signal (`SIGF_DEADTASK`) whenever one of its child threads dies.

Sending a Signal

To send a signal to another task, a task prepares a 32-bit signal mask. It sets the appropriate bit (or bits, if it's sending more than one signal) in the mask to 1 and sets the rest of the bits to 0. For example, if the task wants to send a signal using bit 14, it creates the signal mask "00000000 00000000 01000000 00000000" (in binary). The task then uses the `SendSignal()` call to specify the item number of the task it wants to signal and passes along the signal mask. The kernel logically ORs the signal mask into the receiving task's TCB. Bits set in the `t_SigBits` field of the TCB indicate signals that the task has received, but not yet acknowledged.

Receiving a Signal

A task waits for one or more signals by using the `WaitSignal()` call. The kernel checks to see if any of the bits in the task's signal mask match the bit mask passed `WaitSignal()`, indicating that a signal has been received on that bit. If so, `WaitSignal()` clears the bits that match and immediately returns, letting the task act on any signals it has received. If there are no received signals in the signal mask, the task is put to sleep until it receives a signal it wants.

Freeing Signal Bits

To free up signal bits that a task has allocated, the task uses the `FreeSignal()` call to pass along a *free signal mask*. The free signal mask should have a 1 in each bit where the signal bit is to be freed (that is, set to 0 in the signal mask) and a 0 where the signal bit remains as it is.

Messages

A message is an item that combines three elements: a variable number of bytes of message data, 4 bytes available for an optional reply to the message, and an optional specified place (a reply port, explained later) where a reply to the message can be sent.

A message works with one or two *message ports*: one (required) created by the task receiving the message and another (optional) created by the task sending the message. The message port is an item that sets a user signal bit for incoming message notification. It includes a message queue that receives and stores incoming messages.

Creating a Message

To create a message, a task can use a number of calls including `CreateMsg()`, `CreateSmallMsg()`, and `CreateBufferedMsg()`. These functions accept a string of text as the message's name, a priority for the message, and the item number of the reply port for replies to the message. It returns the item number of the newly created message for working with the message later.

Creating a Message Port

To create a message port, a task uses the `CreateMsgPort()` call, which it provides with a string of text as a message port name. The kernel creates a message queue for the message port, automatically assigns a signal bit for the message port, and gives the message port an item number. The task is now ready to receive messages at the port.

Sending a Message

If a task wants to send a message to another task, it must first know the item number of a message port of the receiving task. (If it knows the name of the message port, it can use the `FindMsgPort()` call to find the item number.) The sending task then uses either `SendMsg()` or `SendSmallMsg()` to fill out a message, providing a destination message port item number and some message data to pass. The kernel inserts the message, according to priority, into the destination port's message queue, then signals the receiving task that a message has arrived at its message port.

Receiving a Message

To receive a message, a task has two options:

- ◆ It can use the `GetMsg()` call to check the message port and retrieve the top message in the list if there are any messages. If there are no messages, the task resumes execution.
- ◆ It can use the `WaitPort()` call to wait for a message. This puts the task into wait state until it receives a message at its message port. The task then retrieves the message and resumes execution.

Replying to a Message

A task that sends a message usually needs a reply from the task that receives the message, so the sending task can specify a message port of its own as the *reply port*. When the receiving task receives the message, it uses either `ReplyMsg()` or `ReplySmallMsg()` to return the same message to the reply port with a 4-byte reply written into the message (stored in the 4-byte `msg_Result` field of the `Message` structure). The sending task receives the reply and reads the 4-byte reply code.

Interpreting a Message

When one task sends a message to another task, the meaning of the message data is completely arbitrary and is determined by the two tasks sharing the message. In many cases, the message data is composed of a pointer to a data structure created in the sending task's memory along with the data structure's size. The receiving task then uses the pointer and size to read the data at that address.

Portfolio Error Codes

Portfolio has a uniform definition for the format of error codes. Whenever system components must return an error, they always use the standard error code format explained here.

All Portfolio error codes are negative 32-bit numbers that are subdivided into multiple subcomponents. Using these components, you can identify which subsystem generated the error, and get a module-specific error number.

Table 1-1 lists the various components of an error code.

Table 1-1 *Error code components.*

Bit(s)	Purpose
31	This bit is always set. It makes all error codes negative numbers.
25-30	Object type that generated the error. This field identifies what generated the error: a folio, a device, a task, or another object type, generated the error. Possible values for this field include ER_FOLI, ER_DEVC, ER_TASK, or ER_LINKLIB.
13-24	Object ID. This is a code that uniquely identifies the component that generated the error. The value for this field is created with <code>MakeErrID()</code> and is basically two 6-bit characters identifying the module that caused the error. For example, kernel errors have an object ID of Kr.
11-12	A severity code. Possible values for this field include: ER_INFO, ER_WARN, ER_SEVERE, or ER_FATAL.
9-10	An environment code. This code defines who created the module that generated the error. Possible error values for this field include: ER_E_SSTM for system modules, ER_E_APPL for application modules, and ER_E_USER for user-code modules.
8	Error class. Possible error values for this field are: either ER_C_STND for a standard error code or ER_C_NSTND for a nonstandard error code. Standard error codes are errors that are well known in the system and have a default string to describe them. Nonstandard errors are module-specific and the module must provide a string to the system to describe the error.
0-7	The actual error code. The meaning of this code depends on the module that generated the error, and whether it is a standard or non-standard error.

The `PrintfSysErr()` kernel call accepts any error code, and prints an error string describing the error to the debugging terminal. When developing code, it is extremely useful to know why a system call is failing.

Tasks and Threads

This chapter provides the background and necessary programming details to create and run tasks and threads. Tasks and threads, although similar, function differently in certain key respects.

This chapter contains the following topics:

Topic	Page Number
What Is a Task?	19
Starting and Ending Tasks	20
Creating a Task	20
Controlling the State of a Task	22
Starting and Ending Threads	24
Advanced Task and Thread Usage	26
Example: Using Threads and Signals	27

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

What Is a Task?

Tasks can be viewed as independent programs, with their own unique memory space. Tasks can spawn other tasks. These spawned tasks (child tasks) quit when the spawning task (parent task) quits, unless the parent task explicitly transfers ownership of the child task to another task before it quits. Although parent and

child tasks are tied together when it's time to quit, they don't share memory. They can't share common data structures, and the child task must allocate its own memory pages, which, if its memory requirements are small, can be wasteful.

Threads are a different kind of spawned or child task. A thread is more tightly bound to its parent task than an ordinary child task is. A thread shares its parent task's memory. The ownership of a thread cannot be transferred to a task that is outside the parent's task family. When the parent dies, each of its threads dies with it because the threads are considered resources of the parent task.

The calls described in this chapter handle both tasks and threads.

Starting and Ending Tasks

A user task on a CD-ROM can start automatically when the CD-ROM starts if the name of the task is LAUNCHME.M2. The operating system loads on boot-up and runs the executable code in LAUNCHME.M2, which starts a task that becomes the child of the shell—a user task.

Creating a Task

Each task is an item in the system. You create a task by calling the `CreateTask()` function.

Each task has a TCB (task control block) associated with it. The TCB data structure is created and maintained by the kernel; a user task can't write to the TCB directly. The settings of the various fields in the TCB are determined by the kind of task (privileged or non-privileged), the parameters passed to the kernel when the task is started, and changes made to the task while it runs. The kernel uses the TCB to allocate and control task resources, memory usage, and so on.

To create a task, you must first load its code from disk using the `OpenModule()` function:

```
Item OpenModule(const char *path, OpenModuleTypes type, const TagArg
               *tags);
```

The first argument specifies the filename of the code you want to load into memory. The second argument specifies whether the code is being loaded in anticipation of being run as a thread or as a task. If you're going to start a task, supply `OPENMODULE_FOR_TASK`, and for a thread `OPENMODULE_FOR_THREAD`. The final argument is reserved for the future and must currently always be `NULL`. The function returns the item number of the loaded code module, or a negative error code for failure.

Once you have an item for a loaded code module, you can use `CreateTask()` to startup a task to execute the module:

```
Item CreateTask(Item module, const char *name, const TagArg *tags);
```

The first argument is the module item that was returned from `OpenModule()`. The second argument is the name that the task will have. You can use this name to find the task later on using `FindTask()`. The final argument is a pointer to an array of `TagArg` structures, which supply extra optional arguments.

The size of the stack used for a task is determined by the stack size value given when the code was linked. If no value was given at link-time, then it can be supplied at run-time using the `CREATETASK_TAG_STACKSIZE` tag.

Setting Priority

When you create a task using `CreateTask()`, the priority of the task being started is determined by the priority value given when the code for the task was linked. If no priority was specified at link-time, then the task will have the same priority as the caller, unless it is overridden by using the `TAG_ITEM_PRI` tag described later.

The Life of a Task

Child tasks created through `CreateTask()` allocate completely separate memory areas for the parent task and its child tasks. Once a child task is launched, it remains in the system as an item until:

- ◆ The child task naturally ends (the program completes its function and dies).
- ◆ The parent task quits (if it still owns the child task).
- ◆ The task that owns the child task explicitly deletes it using `DeleteItem()`.

The launching task remains the owner of the child task throughout the life of the child task unless it passes ownership to another task.

Ending a Task

Typically, a child task ends when its parent task ends. It's good practice to free resources by ending a child task that's no longer needed before the parent task quits. To end a task, call `DeleteTask()`, passing in the task's item number.

```
Err DeleteTask( Item task )
```

Alternatively, a child task can end itself by calling `exit()` or just return:

```
void exit (int status)
```

The function `exit()` never returns.

Note: *The return or exit status can be passed on to the parent task through an exit message if the child was created with `CREATETASK_TAG_MSGFROMCHILD`.*

Deleting the child task lets the kernel clean up all the task's allocated resources, freeing that space for other tasks.

Controlling the State of a Task

A task can be in one of three states at any moment:

- ◆ **Running**, in which case the task is the currently executing task.
- ◆ **Ready to run**, in which case the task is in the ready queue, awaiting execution.
- ◆ **Waiting**, in which case the task is asleep, awaiting an external event to occur, so it can become the running task, or if the priority of the current running task is greater than its own, it moves to the ready queue where it awaits execution.

Each non-privileged task has a priority that ranges from the lowest priority of 10 to the highest priority of 199. Priority determines the order of task execution for tasks that are in the ready queue. The kernel executes only the tasks in the ready queue with the highest-priority values.

The state of a task is determined partly by its own priority, and partly by the other tasks in the system and the interactions among them. A task can wait for other tasks to complete certain processing, at which point it resumes its own processing. Intertask communication is essential because it enables one task to notify a waiting task that an external event is complete.

If there are no external events, preemptive multitasking only takes place when tasks of equal priority are the highest priority tasks in the ready queue. If only one task has the highest-priority, only that task runs, and all other tasks wait until the task finishes or is replaced by another task with higher priority.

Yielding the CPU to Another Task

A task that executes a `Yield()` call cedes its CPU time immediately to another task with the same priority. The `Yield()` call, which neither takes arguments nor returns anything, is:

```
void Yield( void )
```

The next task in the ready queue with equal priority takes the place of the task that executes the `Yield()`. If there is no other task in the ready queue with equal priority, then the task that executes the `Yield()` call immediately resumes processing.

Going To and From Wait State

A task can place itself in wait state with a wait function call, where it uses no CPU time waiting for an event to occur. A wait call defines the event or condition for which the task waits. When the condition occurs, the task moves to the ready queue and resumes running based on its priority. The basic wait call syntax is:

```
int32 WaitSignal( int32 sigMask )
```

Many other functions in Portfolio can make your task wait. Internally, all of these other functions eventually end up calling `WaitSignal()` to put your task to sleep.

`WaitSignal()` is the basic wait call that a task uses to wait for a signal. The wait calls `WaitIO()` and `WaitPort()` are built on `WaitSignal()`. Each of these calls puts a task in the wait state to await a signal, an I/O request return, or a message. Once the signal, I/O request, or message is received, the task returns to the ready queue.

Changing a Task's Parent

Normally, a task created by another task ends when the parent task ends. However, by passing ownership of the child task to another parent task, the child task can live on after the parent task ends. To change ownership, use this call:

```
Err SetItemOwner( Item i, Item newOwner )
```

The `SetItemOwner()` call takes two arguments, the item number of the task whose ownership you wish to change (`Item`), and the item number of the task that is to become the new parent (`newOwner`). This call returns an error code if an error occurs.

A task must be the parent of a child task to change its ownership. If this call is successful, the child task's TCB is removed from the current task's resource table and placed in the parent task's resource table.

Changing Task Priorities

When a task first runs, its priority is set based on a field defined in its TCB data structure. You can change the priority (in the range of 10 through 199) of a task after it has been created by using this call:

```
int32 SetItemPri( Item i, uint8 newpri )
```

The call changes the priority of an existing task. The first argument, `i`, is the item number of the task whose priority you want to change. The second argument, `newpri`, is the new priority value that you want the task to have. The call returns the former priority of the task.

A task can change its own priority by calling:

```
SetItemPri(CURRENTTASKITEM, newPriority)
```

Keep in mind that changing a task's priority affects its status in the ready queue. If you drop the priority below other tasks in the queue, the task can stop executing. If you raise the priority above other tasks in the queue, the task can run alone while the other tasks wait.

Task Quantums

The CPU has a finite amount of processing power. When multiple tasks are competing for CPU time, the kernel allows each task to execute for a predetermined amount of time, and then switches to another task. This process of cycling different tasks is called *round-robin scheduling*. The amount of time that the kernel lets a task run before switching to another task is called a *quantum*.

Round-robin scheduling occurs only when two or more tasks of equal priority are ready to run. If only one task has a given priority, it receives all the CPU's time, as long as no higher-priority tasks are waiting to run.

If a task that is ready to run has a higher priority than the currently running task, the currently running task is immediately put to sleep, and the new task starts executing. This scenario is the essence of a preemptive multitasking system.

When a higher-priority task interrupts a low-priority task, the kernel records the amount of time that the low priority task had left in its quantum. When no higher-priority tasks are ready, control returns to the task that was previously running, and that task gets to complete its time quantum. This procedure is known as quantum remaindering.

It is possible for a task to give up the rest of its time quantum by calling the `Yield()` function. This call allows other ready tasks of equal priority to run immediately. Lower-priority tasks still do not get to run. `Yield()` is very rarely useful, and abusing it can lead to an appreciable decrease in effective system throughput.

Starting and Ending Threads

Threads differ from child tasks in that they are more tightly bound to their parent task. They share the parent task's memory and can't transfer their ownership. However, they still need to be started and ended just as tasks do. This section describes the calls used to start and end threads.

Creating a Thread

The `CreateThread()` function creates a thread:

```
Item CreateThread ( void (*code) (), const char *name,  
                  uint8 pri, int32 stacksize, const TagArg *tags )
```

The first argument, `name`, is the name of the thread to create. The second argument, `pri`, is the priority that the thread will have. The third argument, `code`, is a pointer to the code that the thread will execute. The `stacksize` argument specifies the size in bytes of the thread's stack. Finally, the `tags` argument lets you specify additional optional arguments (discussed later). If successful, the call returns the item number of the thread. If unsuccessful, it returns an error code.

Before calling `CreateThread()`, you must determine what stack size to use for the thread. There is no default size for the `stacksize` argument; however, it's important to make the size large enough so that stack overflow errors do not occur. Stack overflow errors are characterized by random crashes that defy logical analysis, so it's a good approach to start with a large stack size and reduce it until a crash occurs, then double the stack size. In the sample code later in this chapter, the stack size is set to 10000. A good size to start with is 2048.

The priority supplied can be in the range 10 to 199. Specifying a priority of 0 is a special case which cause the thread's priority to equal that of the caller.

Example 2-1 shows the process of creating a thread:

Example 2-1 *Starting a task.*

```
#define STACKSIZE (10000)
...
int main(int argc, char *argv[])
{
    Item T1;
    ...
    T1 = CreateThread( Thread1Proc, "Thread1", 0,
                     STACKSIZE, NULL);
    ...
}

int Thread1Proc()
{
    /* This is the code for Thread1Proc */
}
```

Loaded Threads

`CreateThread()` lets you start a function within your program as a thread. It is also possible to start a loaded code module as a thread using `CreateModuleThread()`:

```
Item CreateModuleThread(Item module, const char *name, const TagArg
                        *tags);
```

The call works in a way very similar to `CreateTask()`, except that a thread is started instead of a task.

Communications Among Tasks

Because threads share memory with the parent task, global variables can pass information among threads and tasks. In the sample code later in this chapter a number of global variables are declared at the start of the program, prior to the code for the threads and the parent task code. As global variables, they are accessible to the main function and to the threads—an example of threads sharing the same memory as the parent task.

The code example given at the end of this chapter is a good illustration of using `CreateThread()` to start two threads. It shows the use of global variables that are shared by all threads, which signal among threads and the parent task. Signals for both threads are first allocated with `AllocSignal()`. The parent task then uses `WaitSignal()` to wait for an event from either of the threads.

Ending a Thread

Use `DeleteThread()` to end a thread when a parent task finishes with it:

```
Err DeleteThread( Item thread )
```

This function takes the item number of the thread to delete as its only argument, and returns a negative error code if an error occurs. Although all threads terminate automatically when the parent task ceases, it's good programming practice to kill a thread as soon as the parent task finishes using it. Note that when you terminate a thread, the thread's memory (which is shared with the parent task) is *not* freed, and remains the property of the parent task. A thread can also terminate itself by calling `exit()` or just returning.

Advanced Task and Thread Usage

The `CreateThread()` and `CreateTask()` functions are convenience functions, which make it easy to create and delete threads for the most common uses. It is sometimes necessary to have better control over thread creation. Both functions take a pointer to an array of `TagArg` structures as parameters. By supplying such an array, you can gain access to the more esoteric features available when creating a task or thread.

You can supply the following tags. Some tags are only valid for either a task or thread, and not both.

- ◆ **TAG_ITEM_PRI.** This is used when spawning either a task or thread. The value can be in the range 10 to 199.
- ◆ **CREATETASK_TAG_ARGC.** This is useful only when spawning a thread. It specifies a 32-bit value that is passed as a first argument to the thread being created. If this value is omitted, the first argument is 0.

- ◆ **CREATETASK_TAG_ARGP.** This is useful only when spawning a thread. It specifies a 32-bit value that is passed as a second argument to the thread being created. If this is omitted, the second argument is 0.
- ◆ **CREATETASK_TAG_CMDSTR.** This is useful only when spawning a task. It specifies a pointer to a string which is used to build a standard C-style argc/argv combination that is given as initial arguments when starting the task.
- ◆ **CREATETASK_TAG_MSGFROMCHILD.** Provides the item number of a message port. The kernel sends a status message to this port whenever the thread or task being created exits. The message is sent by the kernel after the task has been deleted. The `msg_Result` field of the message contains the exit status of the task. This is the value the task provided to `exit()`, or the value returned by the task's primary function. The `msg_Val1` field of the message contains the item number of the task that just terminated. The `msg_Val2` field of the message contains the item number of the task that killed the child task. If a task exited on its own, this is the item number of the task itself. It is the responsibility of the task that receives the status message to delete it when the message is no longer needed by using `DeleteMsg()`.
- ◆ **CREATETASK_TAG_DEFAULTPORT.** Specifying this tag causes a message port to be created for the child task or thread automatically. The item number of the port is put in the `t_DefaultMsgPort` field of the new task structure. This is a convenient way to establish a communication channel with a child quickly.

Example: Using Threads and Signals

Example 2-2 contains sample code for using threads and signals.

The `main()` function launches two threads. These threads simply sit in a loop and count. After a given number of iterations through their loop, they send a signal to the parent task.

When the parent task gets a signal, it wakes up and prints the current counters of the threads to show how much they were able to count.

Example 2-2 *Using threads (signals.c).*

```
/* *****
**
**  @(#) signals.c 95/10/15 1.5
**
** *****/

#include <kernel/types.h>
#include <kernel/task.h>
#include <kernel/operror.h>
#include <stdio.h>

/* *****/

/* Global variables shared by all threads. */
static int32  thread1Sig;
static int32  thread2Sig;
static Item   parentItem;
static uint32 thread1Cnt;
static uint32 thread2Cnt;

/* *****/

/* This routine shared by both threads */
static void DoThread(int32 signal, uint32 amount, uint32 *counter)
{
    uint32 i;

    while (TRUE)
    {
        for (i = 0; i < amount; i++)
        {
            (*counter)++;
            SendSignal(parentItem, signal);
        }
    }
}

/* *****/
```

```
static void Thread1Func(void)
{
    DoThread(thread1Sig, 100000, &thread1Cnt);
}

/*****

static void Thread2Func(void)
{
    DoThread(thread2Sig, 200000,&thread2Cnt);
}

/*****

int main(void)
{
    Item    thread1Item;
    Item    thread2Item;
    uint32 count;
    int32   sigs;

    /* get the item number of the parent task */
    parentItem = CURRENTTASKITEM;

    /* allocate one signal bits for each thread */
    thread1Sig = AllocSignal(0);
    thread2Sig = AllocSignal(0);

    /* spawn two threads that will run in parallel */
    thread1Item = CreateThread(Thread1Func, "Thread1", 0, 2048, NULL);
    thread2Item = CreateThread(Thread2Func, "Thread2", 0, 2048, NULL);

    /* enter a loop until we receive 10 signals */
    count = 0;
    while (count < 10)
    {
        sigs = WaitSignal(thread1Sig | thread2Sig);

        printf("Thread 1 at %d, thread 2 at %d\n",thread1Cnt,thread2Cnt);

        if (sigs & thread1Sig)
            printf("Signal from thread 1\n");
    }
}
```

```
        if (sigs & thread2Sig)
            printf("Signal from thread 2\n");

        count++;
    }

    /* nuke both threads */
    DeleteThread(thread1Item);
    DeleteThread(thread2Item);

    return 0;
}
```

Using Tag Arguments

This chapter provides information on tag arguments (TagArgs).

This chapter contains the following topics:

Topic	Page Number
About Tag Arguments	31
Special Tag Commands	32
How to Specify Tags	33
Parsing Tags	35

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About Tag Arguments

A tag argument (TagArg) is a data structure that is used throughout Portfolio. The structure provides potentially long and variable lists of parameters to system function calls. TagArgs, also referred to simply as tags, provide a very flexible and expandable mechanism that allows Portfolio to evolve with minimal impact on application code.

A TagArg structure contains a command or attribute, and an argument for that command or attribute and looks like this:

```
typedef struct TagArg
{
    uint32  ta_Tag;
    TagData ta_Arg;
} TagArg;
```

TagArg structures are generally used in array form. Many Portfolio function calls take an array of TagArg structures as a parameter. Each function call that uses TagArg arrays defines which commands or attributes can be used with the call.

Arrays of TagArg structures are scanned from start to finish by the function calls that use them. Each element in the array specifies an individual command or attribute pertinent to that function. If a given command appears more than once in a TagArg array, the last occurrence within the array takes precedence. If an unknown command appears in a TagArg, the whole function call fails and returns an appropriate error.

Special Tag Commands

Portfolio provides three special tag commands that have universal meanings in all system calls: TAG_END, TAG_NOP, and TAG_JUMP. By setting the ta_Tag field to these commands, you can control how TagArg arrays are processed by the system.

TAG_END

The TAG_END command tells the system that this structure marks the end of an array of TagArg structures. The system stops scanning the array and goes no further.

TAG_NOP

The TAG_NOP command tells the system to ignore the particular TagArg structure. The system just skips ahead and continues processing with the next TagArg structure.

TAG_JUMP

The TAG_JUMP command links arrays of TagArg structures together. The ta_Arg field for this command must point to another array of TagArg structures. When the system encounters such a TagArg structure, it stops scanning the current array, and resumes scanning at the address specified by ta_Arg.

How to Specify Tags

How Tags are Documented

In this manual and in the *3DO M2 Portfolio Programmer's Reference*, you will often see the tag commands and arguments for a function listed in the form "*ta_Tag (Ta_Arg)*", as shown in Example 3-1.

Example 3-1 *Convention often used to document tag args.*

```
FUNCTIONNAME_TAG_TAGNAME1 (const char *)
```

```
FUNCTIONNAME_TAG_TAGNAME2 (uint32)
```

The tag command is shown in capital letters followed by the data type of the argument in parentheses. This is a convenient way to list tag commands and arguments for a function, but it is important to note that **you never actually specify tags in that format in your code.**

In your code, you specify tag commands and arguments in the form of a tag array or in the form of function arguments using the C language `VarArg` capability.

How to Specify Tags Using a Tag Array

To specify tag arguments using a tag array, define the array and fill in its elements with the desired tag command-argument pairs. You can fill in the command portion of each pair by initialization and the argument portion by assignment. See Example 3-2 below.

Note that you must cast each argument as a `TagData` data type. The `TagData` data type is actually a void pointer (`void *`), and you may see (`void *`) casts used in some older code.

Example 3-2 *Specifying tag arguments in a tag array*

```
static TagArgs tags[] =
{
    {FUNCTIONNAME_TAG_TAGNAME1, 0},
    {FUNCTIONNAME_TAG_TAGNAME2, 0},
    {TAG_END,                      0}
};

uint32 returnValue;
uint32 parm1 = PARM1VALUE;

const char tCharArray[] = "String value";
uint32 tUIntVariable;

tags[0].ta_Arg = (TagData)tCharArray;
tags[1].ta_Arg = (TagData)tUIntVariable;

returnValue = FunctionName(parm1,tags);
```

How to Specify Tags Using VarArg

A superior approach to specifying tag arguments is made possible by using the C VarArg capability.

The VarArg method lets C functions like `printf()` accept a variable number of parameters. You can use the VarArg method to specify tag commands and arguments as normal arguments within the function call you are making. This is generally easier to write, understand, and maintain than tag arrays.

To use the VarArg approach, you must use a form of the function that takes a variable list of arguments. Most Portfolio functions that accept TagArg arrays as parameters also have VarArg counterparts. A VarArg counterpart has the same name as the regular function, but with the addition of a VA suffix.

For example, the `CreateItem()` kernel function accepts an array of TagArg structures as a parameter. The `CreateItemVA()` function is identical in purpose and function, except that it uses a VarArg list of tags instead of a pointer to an array.

To build up a tag list on the stack, you specify the tag commands and their arguments in sequence, separated by commas, and terminate them with a `TAG_END` tag command. You do not have to cast the tag arguments as `(TagData)`.

Example 3-3 shows the same example as Example 3-2, but using `FunctionNameVA()` with a `VarArg` list.

Example 3-3 *Specifying tag arguments in a `VarArg` list*

```
uint32 returnValue;
uint32 parm1 = PARM1VALUE;

const char tCharArray[] = "String value";
uint32 tUIntVariable;

returnValue = FunctionNameVA(parm1,
                             FUNCTIONNAME_TAG_TAGNAME1, tCharArray,
                             FUNCTIONNAME_TAG_TAGNAME2, tUIntVariable,
                             TAGEND);
```

As Example 3-3 shows, the version that calls `FunctionNameVA()` is easier to understand. All the tag commands and their arguments are in the function call itself, instead of in a separate array.

Parsing Tags

If you write utility routines to be shared by many programmers, it is often useful to implement functions that take `TagArg` arrays in a manner similar to the system functions.

The `NextTagArg()` function lets you easily go through all the `TagArg` structures in an array. The function automatically handles all system tag commands like `TAG_NOP` and `TAG_JUMP`, and only returns `TagArg` structures that do not contain system tag commands. You give `NextTagArg()` a pointer to a variable that points to the tag array to process. It returns a pointer to the first `TagArg` structure within the array. You then call the function repeatedly until it returns `NULL`. Every time you call it, it returns the next `TagArg` structure in the array.

The `FindTagArg()` function takes a pointer to an array of `TagArg` structures and a particular tag command. The function scans the supplied array looking for a `TagArg` structure that has the requested command. It returns a pointer to the `TagArg` structure, or `NULL` if no structure with that command is found.

The `GetTagArg()` function works much as `FindTagArg()` does, except that instead of returning a pointer to a `TagArg` structure, it returns the value stored in the `ta_Arg` field of the `TagArg` structure. You also give the function a default data value. If the desired `TagArg` can't be found, the function returns the default data value to you.

Finally, the `DumpTagList()` function displays all the tag commands and arguments in a `TagArg` array to the debugging terminal. This is very useful when you need to see every tag value being passed to a function call.

Managing Linked Lists

This chapter explains how to manage linked lists.

This chapter contains the following topics:

Topic	Page Number
About Linked Lists	38
Creating and Initializing a List	39
Adding a Node to a List	40
Changing the Priority of a Node	42
Removing a Node From a List	42
Finding Out If a List Is Empty	43
Traversing a List	43
Finding a Node by Name	45
Finding a Node by Its Ordinal Position	45
Determining the Ordinal Position of a Node	46
Counting the Number of Nodes in a List	46
Primary Data Structures	46

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

About Linked Lists

Linked lists are used throughout Portfolio and in applications you create with it. To make using lists easier (and to meet the needs of system software and its lists), the kernel defines a special type of linked list, known as a Portfolio list. The kernel also provides a variety of functions for creating and managing Portfolio lists.

Like all linked lists, Portfolio lists are dynamic; they can expand and contract as needed. Their contents, known as nodes, are ordered (there is a first node, a second node, and so on), and you can add a new node at any position in a list. The following sections explain how Portfolio lists are different from other linked lists.

Characteristics of Portfolio Lists

Unlike ordinary linked lists, which contain only nodes, Portfolio lists also contain a special component known as an anchor, which marks both ends of the list. The anchor (which is implemented as a C union) serves as both the beginning-of-list marker (known as the head anchor) and the end-of-list marker (known as the tail anchor). Figure 3-1 illustrates an anchored list.

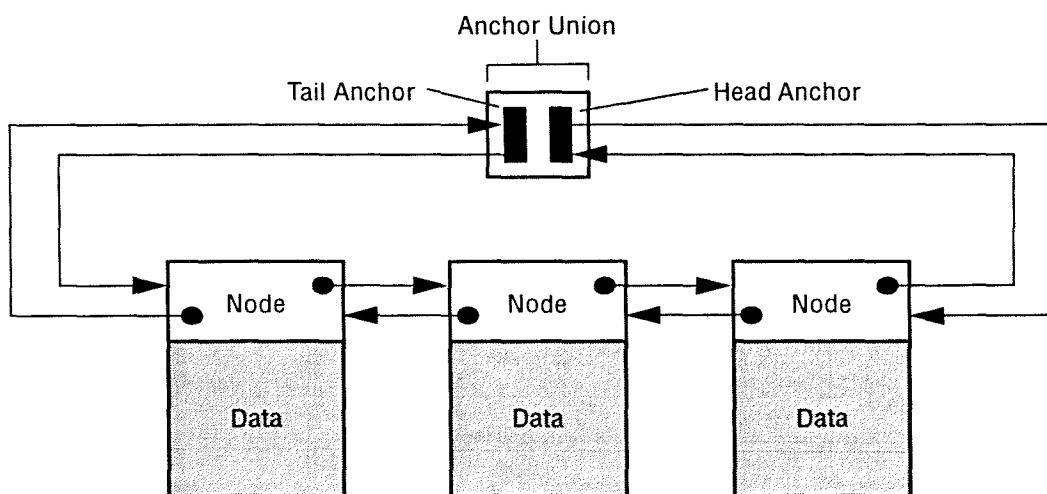


Figure 4-1 *Anchored list.*

When a task traverses a Portfolio list, it determines whether it's at the beginning or end of the list by testing to see if the subsequent node is an anchor.

As the previous illustration shows, Portfolio lists are doubly linked: Each node contains two pointers, one to point to the following node or anchor and one to the previous node or anchor. As a result, back-to-front list traversals are as efficient as front-to-back traversals.

Characteristics of Nodes

In Portfolio lists, there are special data structures that contain only information needed for list management. This information includes the necessary forward and backward pointers, a priority value (described later in this section), and other fields that are used primarily by the operating system. A node can also have a name.

To create a list component, you define a data structure whose first field is a node. An example is the `NoteTracker` structure defined in the music library:

Example 4-1 *The Note Tracker structure in the music library.*

```
typedef struct NoteTracker
{
    Node      nttr_Node;
    int8      nttr_Note;
    int8      nttr_MixerChannel;
    uint8     nttr_Flags;
    int8      nttr_Channel;      /* MIDI */
    Item      nttr_Instrument;
} NoteTracker;
```

To pass such a component to one of the many list-manipulation functions that takes a `Node` structure as an argument, you simply cast the argument to type `Node`. Here's an example:

```
AddTail( &DSPPData.dspp_ExternalList, (Node *) dext )
```

Every node in a list has a priority (a value from 0–255 that is stored in the `n_Priority` field of the `Node` structure). When you use a list, you have the option of keeping its nodes sorted by priority (done automatically by the kernel if you use `InsertNodeFromHead()` or `InsertNodeFromTail()`), or you can specify other ways to arrange the contents (by using the `UniversalInsertNode()` function). You can also change the priority of a node in a list with the `SetNodePri()` function, whereupon the kernel automatically repositions the node in the list to reflect its new priority value.

A node can be in only one list at a time.

Creating and Initializing a List

To create an empty linked list, you first create a variable of type `List`. You then prepare the list, which initializes its head and tail anchors, by calling the `PrepList()` function:

```
void PrepList( List *l)
```

Another way to initialize a list is by using the `PREPLIST()` macro. This macro lets you define a fully initialized `List` as a variable. By using the macro, you avoid the need to call the `PrepList()` function. Here is an example of using the `PREPLIST` macro to create a variable called `listOfStuff`:

```
static List listOfStuff = PREPLIST(listOfStuff)
```

Adding a Node to a List

You can add a node to a list in any of the following ways:

- ◆ The beginning or end of a list.
- ◆ A position in the list that corresponds to the node's priority.
- ◆ A position in the list determined by comparing one or more values contained in the node to values of nodes currently in the list, or relative to another node already in the list.
- ◆ Before or after another node already in the list.

The following sections describe each of these cases.

Adding a Node to the Head of a List

To add a node to the head of a list, use the `AddHead()` function:

```
void AddHead( List *l, Node *n )
```

The `l` argument is a pointer to the list to which to add the node, the `n` argument is a pointer to the node to add.

Adding a Node to the Tail of a List

To add a node to the tail of a list, use the `AddTail()` function:

```
void AddTail( List *l, Node *n )
```

The `l` argument is a pointer to the list to which to add the node, the `n` argument is a pointer to the node to add.

Adding a Node After Another Node in a List

To add a node to a list and position it after another node already in the list, use the `InsertNodeAfter()` function:

```
void InsertNodeAfter( Node *oldNode, Node *newNode )
```

The `oldNode` argument is a pointer to a node already in the list, the `newNode` argument is a pointer to the new node to insert.

Adding a Node Before Another Node in a List

To add a node to a list and position it before another node already in the list, use the `InsertNodeBefore()` function:

```
void InsertNodeBefore( Node *oldNode, Node *newNode )
```

The `oldNode` argument is a pointer to a node already in the list, while the `newNode` argument is a pointer to the new node to insert.

Adding a Node According to Its Priority

The nodes in a list are often arranged by priority. To insert a new node immediately before any other nodes of the same priority, use the `InsertNodeFromHead()` function:

```
void InsertNodeFromHead( List *l, Node *n )
```

As in the other functions for adding nodes, the `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add.

The name `InsertNodeFromHead()` refers to the way the kernel traverses the list to find the correct position for a new node: it compares the priority of the new node to the priorities of nodes already in the list, beginning at the head of the list, and inserts the new node immediately after nodes with higher priorities. If the priorities of all the nodes in the list are higher than the priority of the new node, the node is added to the end of the list.

To insert a new node immediately after all other nodes of the same priority, you use the `InsertNodeFromTail()` function:

```
void InsertNodeFromTail( List *l, Node *n )
```

The `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add. As with `InsertNodeFromHead()`, the name `InsertNodeFromTail()` refers to the way the kernel traverses the list to find the correct position for the new node: it compares the priority of the new node to the priorities of nodes already in the list, beginning at the tail of the list, and inserts the new node immediately before nodes with lower priorities. If the priorities of all the nodes in the list are lower, the node is added to the head of the list.

Adding a Node in Alphabetical Order

The `InsertNodeAlpha()` function inserts a node into a list according to alphabetical order of the name of the node. The list is assumed to be already sorted in alphabetical order.

The syntax of the `InsertNodeAlpha()` function is:

```
void InsertNodeAlpha(List *l, Node *n);
```

Adding a Node According to Other Node Values

Arranging list nodes by priority is only one way to order a list. You can arrange the nodes in a list by node values other than priority by using the `UniversalInsertNode()` function to insert new nodes:

```
void UniversalInsertNode( List *l, Node *n, bool (*f) (Node *n,
Node *m) )
```

The `l` argument is a pointer to the list to which to add the node, while the `n` argument is a pointer to the node to add. Like `InsertNodeFromHead()`, `UniversalInsertNode()` compares the node to be inserted with nodes already in the list, beginning with the first node. The difference is that it uses the comparison function `f` provided by your task to compare the new node to existing nodes. If the comparison function returns `TRUE`, the new node is inserted immediately before the node to which it was compared. If the comparison function always returns `FALSE`, the new node becomes the last node in the list. The comparison function, whose arguments are pointers to two nodes, can use any data in the nodes for the comparison.

Changing the Priority of a Node

List nodes are often ordered by priority. You can change the priority of a list node (and thereby change its position in the list) by using the `SetNodePri()` function:

```
uint8 SetNodePri( Node *n, uint8 newpri )
```

The `n` argument is a pointer to the list node whose priority you want to change. The `newpri` argument specifies the new priority for the node (a value from 0 to 255). The function returns the previous priority of the node. When you change the priority of a node, the kernel automatically moves the node immediately.

Removing a Node From a List

You can remove the first node, the last node, or a specific node from a list. The following sections explain how to do it.

Removing the First Node

To remove the first node from a list, use the `RemHead()` function:

```
Node *RemHead( List *l )
```

The `l` argument is a pointer to the list from which you want to remove the node. The function returns a pointer to the node that was removed from the list or `NULL` if the list was empty.

Removing the Last Node

To remove the last node from a list, use the `RemTail()` function:

```
Node *RemTail( List *l )
```

The `l` argument is a pointer to the list from which you want to remove the node. The function returns a pointer to the node that was removed from the list or `NULL` if the list was empty.

Removing a Specific Node

To remove a specific node from a list, use the `RemNode()` function:

```
void RemNode( Node *n )
```

The `n` argument is a pointer to the node you want to remove. Because a node can be only in one list, the node is automatically removed from the correct list.

Finding Out If a List Is Empty

To find out if a list is empty, use `IsListEmpty()`:

```
bool IsListEmpty( const List *l )
```

The `l` argument is a pointer to the list to check. The macro returns `TRUE` if the list is empty or `FALSE` if it isn't.

Traversing a List

You can traverse a list equally quickly from front to back or from back to front. The following sections explain how.

Traversing a List From Front to Back

To traverse a list from front to back, use `ScanList()`. It iterates through all the elements in the list.

Example 4-2 *Traversing a list front to back.*

```
ScanList(list,n,DataType)
{
    /* here you can dereference "n" */
}
```

`ScanList()` is a macro made up of the simpler macros `FirstNode()`, `IsNode()`, and `NextNode()`.

Use `FirstNode()` to get the first node in a list:


```
Node *FirstNode( const List *l )
```

The *l* argument is a pointer to the list the node is in. The macro returns a pointer to the first node in the list or, if the list is empty, a pointer to the tail anchor.

To check to see if a node is an actual node rather than the tail anchor, use `IsNode()`:

```
bool IsNode( const List *l, const Node *n )
```

The *l* argument is a pointer to the list containing the node; the *n* argument is a pointer to the node to check. The macro returns FALSE if it is the tail anchor or TRUE for any other node. (`IsNode()` returns TRUE for any node that is not the tail anchor, no matter if the node is in the specified list.)

To go from one node to its successor in the same list, use `NextNode()`:

```
Node *NextNode( const Node *n )
```

The *n* argument is a pointer to the current node. (This node must be in a list.) The macro returns a pointer to the next node in the list or, if the current node is the last node in the list, to the tail anchor.

Traversing a List From Back to Front

To traverse a list from front to back, use the `ScanListB()` macro. It iterates through all the elements in the list.

Example 4-3 *Traversing a list back to front.*

```
ScanListB(list,n,DataType)
{
    /* here you can dereference "n" */
}
```

`ScanListB()` is a macro made up of the simpler macros `LastNode()`, `IsNodeB()`, and `PrevNode()`.

Use `LastNode()` to get the last node in a list:

```
Node *LastNode( const List *l )
```

The *l* argument is a pointer to the list the node is in. The macro returns a pointer to the last node in the list or, if the list is empty, a pointer to the head anchor.

To check to see if a node is an actual node rather than the head anchor, use `IsNodeB()`:

```
bool IsNodeB( const List *l, const Node *n )
```

The `l` argument is a pointer to the list containing the node to check; the `n` argument is a pointer to the node to check. The macro returns `FALSE` if it is the head anchor or `TRUE` for any other node. (The macro returns `TRUE` for any node that is not the head anchor, whether or not the node is in the specified list.)

To go from one node to its predecessor in the list, use the `PrevNode()` macro:

```
Node *PrevNode( const Node *n )
```

The `n` argument is a pointer to the current node. (This node must be in a list.) The macro returns a pointer to the previous node in the list or, if the current node is the first node in the list, to the head anchor.

Finding a Node by Name

Because list nodes can have names, you can search for a node with a particular name with the `FindNamedNode()` function:

```
Node *FindNamedNode( const List *l, const char *name )
```

The `l` argument is a pointer to the list to search; the `name` argument is the name of the node for which to search. The function returns a pointer to the `Node` structure or `NULL` if the named node is not found. The search is not case-sensitive; that is, the kernel does not distinguish uppercase and lowercase letters in the node names.

Finding a Node by Its Ordinal Position

You can find a node that appears in a given position from the beginning or end of a list. To find a node from the beginning of a list, use the `FindNodeFromHead()` function:

```
Node *FindNodeFromHead( const List *l, uint32 position )
```

The `l` argument is a pointer to the list to search; the `position` argument is the position of the node sought within the list, counting from the head of the list. The first node in the list has position 0. The function returns a pointer to the node, or `NULL` if there are not enough nodes in the list for the requested position.

To find a node from the end of a list, use the `FindNodeFromTail()` function:

```
Node *FindNodeFromTail( const List *l, uint32 position )
```

The `l` argument is a pointer to the list to search; the `position` argument is the position of the node sought within the list, counting from the tail of the list. The last node in the list has position 0. The function returns a pointer to the node, or `NULL` if there are not enough nodes in the list for the requested position.

Determining the Ordinal Position of a Node

Given a list and a node, you can determine the position of the node within the list, counting from the beginning or the end of the list. To determine the position of a node relative to the beginning of a list, use the `GetNodePosFromHead()` function:

```
int32 GetNodePosFromHead( const List *l, const Node *n )
```

The `l` argument is a pointer to the list in which the node is located; the `n` argument is the node to find in the list. The function scans the list looking for the node, and returns the position of the node within the list. The first node in the list has position 0. If the node cannot be located in the list, the function returns -1.

To determine the position of a node relative to the end of a list, use the `GetNodePosFromTail()` function:

```
int32 GetNodePosFromTail( const List *l, const Node *n )
```

The `l` argument is a pointer to the list in which the node is located; the `n` argument is the node to find in the list. The function scans the list looking for the node, and returns the position of the node relative to the end of the list. The last node in the list has position 0. If the node cannot be located in the list, the function returns -1.

Counting the Number of Nodes in a List

To count the number of nodes in a list, use the `GetNodeCount()` function:

```
uint32 GetNodeCount( const List *l )
```

The `l` argument is a pointer to the list of which you want count the nodes. The function returns the number of nodes currently in the list.

Primary Data Structures

The following sections describe the most important data structures for working with linked lists.

The Node Structure

The Node data structure is the standard structure for any named node in a linked list. The `n_Name` field lets you easily locate any node in a linked list.

Example 4-4 *The Node data structure.*

```

typedef struct Node
{
    struct Node  *n_Next;           /* pointer to next node in list */
    struct Node  *n_Prev;           /* pointer to previous node in list */
    uint8        n_SubsysType;      /* what folio manages this node */
    uint8        n_Type;            /* what type of node for the folio */
    uint8        n_Priority;        /* queueing priority */
    uint8        n_Flags;           /* flags used by the system */
    int32        n_Size;            /* total size of node including hdr */
    char         *n_Name;           /* ptr to null terminated string or NULL */
} Node, *NodeP;

```

MinNode and NamelessNode Structures

In addition to the regular Node structure, Portfolio also defines the MinNode and NamelessNode structures. These structures can be used in place of the full Node structure when defining your own lists. These structures have much less overhead than the Node structure, so your lists use less memory.

The NamelessNode structure is identical to the Node structure, except that it doesn't have the n_Name field. You can use the NamelessNode structure in place of a Node structure for all the functions and macros explained in this chapter, except for the FindNamedNode(), DumpNode(), and InsertNodeAlpha() functions. Since these functions use the n_Name field, they cannot handle the NamelessNode.

Example 4-5 *The NamelessNode structure.*

```

/* Node structure used when the Name is not needed */
typedef struct NamelessNode
{
    struct NamelessNode *n_Next;
    struct NamelessNode *n_Prev;
    uint8 n_SubsysType;
    uint8 n_Type;
    uint8 n_Priority;
    uint8 n_Flags;
    int32 n_Size;
} NamelessNode, *NamelessNodeP;

```

The `MinNode` structure is very small, and provides just enough information to link within lists. It can be used with most functions and macros explained in this chapter, except for `SetNodePri()`, `InsertNodeFromTail()`, `InsertNodeFromHead()`, `InsertNodeAlpha()`, `FindNamedNode()`, and `DumpNode()`. These functions use the extra fields found in the `Node` structure, and cannot work with the simple `MinNode` structure.

Example 4-6 *The MinNode structure.*

```
/* Node structure used for linking only */
typedef struct MinNode
{
    struct MinNode *n_Next;
    struct MinNode *n_Prev;
} MinNode;
```

The List Data Structure

The `List` data structure is the means by which nodes are linked together.

Example 4-7 *The List data structure.*

```
typedef struct List
{
    uint32 l_Flags;
    ListAnchor ListAnchor; /* Anchor point for list of nodes */
} List, *ListP;
```

The ListAnchor Union

The `ListAnchor` union contains the forward and backward pointers for the first and last node of any linked list.

Example 4-8 *The ListAnchor union.*

```
typedef union ListAnchor
{
    struct
    {
        Link links;
        Link *filler;
    } head;
    struct
    {
        Link *filler;
        Link links;
    } tail;
} ListAnchor;
```

The Link Data Structure

The Link data structure contains the forward and backward pointers for a linked list.

Example 4-9 *The Link data structure.*

```
typedef struct Link
{
    struct Link *fblink;
    struct Link *blink;
} Link;
```

Managing Memory

All tasks need memory. This chapter explains how to allocate memory, how to share it with other tasks, how to get information about it, and how to free it.

This chapter contains the following topics:

Topic	Page Number
About Memory	52
Managing Memory	53
Getting Information About Memory	55
Allowing Other Tasks to Write to Your Memory	57
Transferring Memory to Other Tasks	58
Interfacing to the System's Free Page List Directly	58
Debugging Memory Usage	59
Example: Allocating and Deallocating Memory	61
Other Memory Topics (Caches, Bit Arrays)	62

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

About Memory

On 3DO systems, DRAM is generic memory available for use by the CPU and standard DMA devices. Current consumer 3DO M2 systems include a minimum of 8 MB of DRAM; development systems may have much more. Future consumer systems may also have larger memory capacities.

How Memory Works

The following sections explain how memory is arranged and how it gets allocated and freed.

Free Page List

In the 3DO system the hardware divides memory into pages. The current size of a memory page is 4 KB. In a system with 8 MB of DRAM, that means there are 2048 pages of memory.

The kernel maintains a list of available pages in the system. If a given page is not on the free page list, it means it has been allocated by a task or by Portfolio itself for system use. When a task allocates a page of memory, it becomes the owner of that page.

Each task in the 3DO environment owns a certain number of memory pages. Tasks can write only to pages of memory they own, or pages of memory for which they were explicitly granted write-access by other tasks.

It is important to understand that threads always have the same access privileges as their parent task. From the memory manager's standpoint, a task and its threads are all one and the same.

Free Memory Lists

Each task in the system has a free memory list. This list is used to keep track of the memory usage within the pages of memory that a task owns. The list is made up of a collection of all the blocks of memory that have not been allocated within the pages owned by the task.

In addition to the free memory list for tasks, the kernel maintains a supervisor memory list, which is used to track allocations made by Portfolio's privileged code. The memory for items comes out of this free memory list.

Getting Pages In and Out of a Free Memory List

When a task allocates a block of memory using the `AllocMem()` function, the kernel attempts to find a suitable range of unused memory in the task's free memory list. If it cannot find any, the kernel then checks to see if there are enough

pages available in the system's free page list to satisfy the allocation. If there are, the kernel allocates these pages for the task and gives it ownership. If there aren't enough free pages in the free page list, then the allocation request fails.

Once you are done with a block of memory, you can return it to your task's free memory list by calling the `FreeMem()` function. This makes the memory available for subsequent allocations by your task or threads of your task.

Freeing memory to a task's free list doesn't make the memory available to other tasks in the system. This is because the pages of memory used for the allocation still remain the property of the task. In order for other tasks to be able to allocate these pages, they must be returned to the system's free page list. The `ScavengeMem()` function will transfer any totally unused pages of memory from the task's free memory list to the system's free page list, making the pages available to others tasks in the system.

Managing Memory

The following sections explain in more detail how to manage memory.

Allocating a Block of Memory

The standard way to allocate memory is with the `AllocMem()` function:

```
void *AllocMem( int32 memSize, uint32 memFlags)
```

The `memSize` argument specifies the size of the memory block to allocate, in bytes. The `memFlags` argument contains memory allocation flags that specify options for the memory allocator. The function returns a pointer to the allocated block of memory or `NULL` if the block couldn't be allocated.

Memory Allocation Flags

In the `memFlags` argument, you must include one of the following flags to specify the type of memory to allocate:

- ◆ **MEMTYPE_NORMAL.** Allocates regular memory. This is currently the only type of memory available. This flag must therefore be specified with any memory allocation that is made.

You can use the following optional flags to specify some options to the memory allocator:

- ◆ **MEMTYPE_FILL.** Sets every byte in the memory block to the value of the lower-8 bits of the flags. If this flag is not set, the previous contents of the memory block are not changed. Using this flag is slower than not using it, so don't set this flag if you plan to initialize memory with your own data.
- ◆ **MEMTYPE_TRACKSIZE** Tells the system to automatically keep track of the size of the allocated block of memory. When you later free the block of

memory, you then do not need to explicitly specify the size of the memory block. Using this option increases the size of the allocation slightly in order to keep track of the size.

It is sometimes necessary to allocate a block of memory which is aligned to a fixed boundary in memory. Many DMA devices require particular alignment for optimal functioning, or to work at all. The `AllocMemAligned()` function works like `AllocMem()`, except that it takes an extra parameter indicates the minimal alignment needed for the allocation.

```
void *AllocMemAligned( int32 memSize, uint32 memFlags, uint32
                      alignment )
```

The address returned by the function is either `NULL`, for failure, or a pointer to a memory block whose address is an even multiple of the supplied alignment parameter.

Freeing a Block of Memory

To free a memory block allocated with `AllocMem()`, use `FreeMem()`:

```
void FreeMem( void *mem, int32 memSize )
```

The `mem` argument points to the memory block to free. The `memSize` argument specifies the size in bytes of the block to free. The size you specify must always be the same as the size specified when you allocated the block with one exception. If the memory was allocated using the `MEMTYPE_TRACKSIZE` flag, then you must supply the constant `TRACKED_SIZE` as the size.

Reallocating a Block of Memory

It is sometimes useful to change the size of a previously allocated block of memory. For example, you may wish to allocate a larger than needed block of memory in order to perform DMA to it using a device requiring a minimal block size (the CD-ROM for example). Once the DMA operation is complete, you can resize the memory block to be only as large as the amount of useful data you obtained. To resize a block of memory, you use `ReallocMem()`:

```
void *ReallocMem( void *mem, int32 oldSize, int32 newSize, uint32
                 memFlags )
```

The `mem` argument is a pointer to the block of memory to resize. The `oldSize` argument specifies the old size of the block of memory to resize. If the block of memory was originally allocated with `MEMTYPE_TRACKSIZE`, then the `oldSize` argument should be set to `TRACKED_SIZE`. The `newSize` argument specifies the new size that the block of memory should have. Finally, the `memFlags` argument specifies some options for the allocation. The function returns a pointer to the new block of memory. This may be the same address as the old block, or it may be different.

At present time, the `ReallocMem()` function has limited capabilities. It cannot currently make a block of memory larger, only smaller. In addition, if the old memory block was allocated with `MEMTYPE_TRACKSIZE`, then the new memory block must also be. Finally, the `MEMTYPE_FILL` option is not supported, since it only makes sense when making a block of memory larger.

In general, it is better to completely free a block of memory and allocate a whole new one unless you have some data in the buffer which you do not want to lose. When abused, `ReallocMem()` can lead to memory fragmentation.

Getting Information About Memory

The following sections explain how to find out how much memory is available, and other useful information.

Finding Out How Big a Block of Memory Is

If you allocate a block of memory using the `MEMTYPE_TRACKSIZE` flag, you do not need to keep track of the size of the block in order to be able to free it, the system keeps track of the size for you. You can use the `GetMemTrackSize()` function to obtain the store size of a given block of memory:

```
int32 GetMemTrackSize( const void *mem )
```

Finding Out How Much Memory Is Available

You can find out how much memory is available by calling `GetMemInfo()`:

```
void GetMemInfo( MemInfo *minfo, uint32 infoSize, uint32 memflags )
```

The `memflags` argument specifies the type of memory you want to find out about. This value must currently always be `MEMTYPE_NORMAL`.

The information about available memory is returned in a `MemInfo` structure pointed to by the `minfo` argument:

```
typedef struct MemInfo
{
    uint32 minfo_TaskAllocatedPages;
    uint32 minfo_TaskAllocatedBytes;
    uint32 minfo_FreePages;
    uint32 minfo_LargestFreePageSpan;
    uint32 minfo_SystemAllocatedPages;
    uint32 minfo_SystemAllocatedBytes;
    uint32 minfo_OtherAllocatedPages;
} MemInfo;
```

This structure contains the following information:

- ◆ **minfo_TaskAllocatedPages.** Specifies the number of pages the current task owns.
- ◆ **minfo_TaskAllocatedBytes.** Specifies the number of bytes allocated by the current task out of its allocated pages.
- ◆ **minfo_SystemAllocatedPages.** Specifies the number of pages allocated by the system to store items and other system-private allocations.
- ◆ **minfo_FreePages.** Specifies the number of unallocated memory pages in the system.
- ◆ **minfo_LargestFreePageSpan.** Specifies the largest number of contiguous unallocated pages in the system.
- ◆ **minfo_SystemAllocatedPages.** Specifies the number of pages allocated by the system to store items and other system-private allocations.
- ◆ **minfo_SystemAllocatedBytes.** Specifies the number of bytes allocated by the system to store items and other system-private allocations.
- ◆ **minfo_OtherAllocatedPages.** Specifies the number of pages allocated by tasks other than the current task.

Warning: Do not depend on these numbers, as they may change if other tasks in the system allocate memory at the same time you are calling *GetMemInfo()*.

Getting the Size of a Memory Page

You can get the size of a memory page by calling the *GetPageSize()* function:

```
int32 GetPageSize( uint32 memFlags )
```

The *memFlags* argument specifies the type of memory you are interested in. This must currently always be *MEMTYPE_NORMAL*. The return value contains the size of the page for the specified memory type, in bytes.

Validating Memory Pointers

When working with memory, sometimes it is necessary to verify the validity of memory pointers. This can be extremely useful when debugging, for example, in *ASSERT()* macros. Portfolio validates memory pointers that are passed to it, and rejects any corrupt ones.

The *IsMemReadable()*, *IsMemWritable()*, and *IsMemOwned()* functions let your code verify the validity of pointers.

```
bool IsMemReadable(const void *mem, int32 memSize)
bool IsMemWritable(const void *mem, int32 memSize)
```

```
bool IsMemOwned(const void *mem, int32 memSize)
```

You supply a pointer to a memory region and the size of that region. The functions return TRUE if the current task or thread can access the complete memory region.

Allowing Other Tasks to Write to Your Memory

When a task owns memory, other tasks cannot write to that memory unless the owner gives them permission. To give another task permission to write to one or more of your memory pages, or to revoke write permission that was previously granted, call `ControlMem()`:

```
Err ControlMem( void *mem, int32 memSize, ControlMemCmds cmd, Item
               task )
```

Each page of memory has a control status that specifies which task owns the memory and which tasks can write to it. Calls to `ControlMem()` change the control status for entire pages. The `mem` argument, a pointer to a memory location, and the `memSize` argument, the amount of the contiguous memory, in bytes, beginning at the memory location specified by the `mem` argument, specify which memory control status to change. If the `mem` and `memSize` arguments specify any part of a page, changes are made to the entire page. The `task` argument specifies which task to change the control status for. You can make multiple calls to `ControlMem()` to change the control status for more than one task. The `cmd` argument is a constant that specifies the change to be made to the control status. The possible values are given below:

- ◆ **MEMC_OKWRITE.** Grants permission to write to the memory of the task specified by the `task` argument. A task argument value of 0 makes the range of memory writable by all tasks.
- ◆ **MEMC_NOWRITE.** Revokes permission to write to the memory from the task specified by the `task` argument. A task argument value of 0 revokes write permission for all tasks.

(One other possible value, `MEMC_GIVE`, is described in the next section.)

For a task to change the control status of memory pages with `ControlMem()`, it must own that memory, with one exception: a task that has write access to memory it doesn't own can relinquish its write access using `MEMC_NOWRITE` as the value of the `cmd` argument.

A task can use `ControlMem()` to prevent itself from writing to memory that it owns, which is useful during debugging.

Transferring Memory to Other Tasks

The previous section explained how to use `ControlMem()` to grant and revoke write access to memory pages. You can also use `ControlMem()` to transfer ownership of memory pages to other tasks:

```
Err ControlMem( void *mem, int32 memSize, ControlMemCmds cmd, Item
               task )
```

If the value of the `cmd` argument is `MEMC_GIVE`, the call gives the memory pages to the task specified by the `task` argument. In this case, the `mem` argument, a pointer to a memory location, and the `memSize` argument, the amount of the contiguous memory, in bytes, beginning at the memory location specified by the `mem` argument, together specify the memory to give away. If these two arguments specify any part of a page, the entire page is given away.

You can also give memory pages back to the kernel — and thereby return them to the system-wide free memory pool— by calling `ControlMem()` as described here, but with zero as the value of the `task` argument. Memory pages can be returned to the kernel automatically when a task calls the `ScavengeMem()` function described under the `ScavengeMem()` entry in *3D0 System Programmer's Reference*.

Interfacing to the System's Free Page List Directly

When there is insufficient memory in a task's free memory list to allocate a block of memory, the kernel automatically provides additional memory pages from the system's free page list. Tasks can also get pages directly from the system's free page list instead of relying on the kernel to do it by calling the `AllocMemPages()` function:

```
void *AllocMemPages( int32 memSize, uint32 memFlags )
```

The `memSize` argument specifies the amount of memory to transfer, in bytes. If the specified size is not a multiple of the page size for the type of memory requested, the kernel transfers enough full pages to satisfy the request.

The `memFlags` argument specifies the type of memory to allocate. This must currently always be `MEMTYPE_NORMAL`.

To return allocated pages to the system's free page list, use `FreeMemPages()`;

```
void FreeMemPages( void *mem, int32 memSize )
```

The `mem` argument is a pointer to the block of memory that was allocated with `AllocMemPages()`. The `memSize` argument corresponds to the size provided to `AllocMemPages()` when the memory was first allocated.

Allocating and freeing pages of memory yourself is faster than using `AllocMem()` and `FreeMem()`, but it doesn't have the convenience of giving you exactly the amount of memory you need, and only gives you relatively large pages instead.

Debugging Memory Usage

Problems can occur when your application allocates or manages memory if the memory is misused or isn't freed up after use. The kernel provides a memory debugging mode, known as `MemDebug`, to help spot memory management problems. To use `MemDebug`, do the following:

1. Add `CreateMemDebug()` as the first instruction in the `main()` routine of your program:

```
Err CreateMemDebug(const TagArg *args)
```

The only argument to the function must currently always be `NULL`. The call returns ≥ 0 if `MemDebug` was created, or a negative failure code if an error occurred.

2. Add `ControlMemDebug()` as the second instruction in the `main()` routine of your program:

```
Err ControlMemDebug(uint32 controlFlags)
```

`controlFlags` is a set of bit flags that control various options of the memory debugging code. See "Chapter 1, Kernel Calls," in the *3DO System Programmer's Reference* for a description of each flag.

3. Add `DumpMemDebug()` and `DeleteMemDebug()` as the last instructions in the `main()` routine of your program.

```
Err DumpMemDebug(const TagArg *args)
```

```
Err DeleteMemDebug(void)
```

`args` is a pointer to an array of tag arguments containing extra data for this function.

4. Recompile your entire application with `MEMDEBUG` defined on the ARM compiler's command line. For the DCC compiler, this is done by adding

```
-DMEMDEBUG
```

to the compile line.

Other than `CreateMemDebug()`, `ControlMemDebug()`, `DumpMemDebug()`, and `DeleteMemDebug()`, there is no difference in the way your program works. When the program allocates memory, each memory allocation is tracked and managed. In addition, `MemDebug` makes sure that illegal or dangerous memory

use is detected and flagged. When the program exits, any memory left allocated is displayed, along with the line number and source file from where memory was allocated.

When all the control flags are turned on, MemDebug checks and reports the following problems:

- ◆ Memory allocations of 0.
- ◆ Memory freed with a NULL or bogus memory pointer.
- ◆ Memory freed of a size that does not match the size of memory that was allocated.
- ◆ Cookies on either side of all memory allocations are checked so they are not altered from the time a memory allocation is made to the time the memory is released. A change in the cookies indicates that your program is writing beyond the bounds of allocated memory.

When MEMDEBUG is defined during compilation, all standard memory allocation calls are automatically vectored through special versions of the memory allocation routines that track the source file and source line where the function call originated from. When MemDebug reports an error, it will print out the name of the source file and line number when possible. If a source module wasn't recompiled with MemDebug, or if memory allocations are done by folios on your task's behalf, then MemDebug will still track the allocations and do all of its checking, but it will not be able to display source file and source line information when it reports a problem.

If you call `DumpMemDebug()` at any time within your application, you can get a detailed listing of all memory currently allocated, showing the source line and source file from which the allocation occurred. This function also outputs statistics about general memory allocation patterns including: the number of memory allocation calls that have been performed, the maximum number of bytes allocated at any one time, current amount of allocated memory, and so on. All of this information is displayed on a per-thread basis, as well as globally for all threads. When using link libraries that haven't been recompiled with MEMDEBUG defined, the memory debugging subsystem will still can track the allocations, but will not report the source file or line number where the error occurred. It reports `<unknown source file>` instead.

Two additional calls, `SanityCheckMemDebug()` and `RationMemDebug()` provide you with more memory debugging capabilities.

`SanityCheckMemDebug()` checks outstanding memory allocations and checks all cookies to see if they have been corrupted. For example, if you want to find the location of a chunk of memory that is being corrupted, you could place

`SanityCheckMemDebug()` calls throughout the program. Look at the return data of each call until you locate the place in the program where the cookies are corrupted.

`RationMemDebug()` lets you deliberately cause memory allocation failures under specified conditions in order to test error handling and recovery paths. You can place `RationMemDebug()` calls throughout your program with tags that turn memory rationing (i.e., failure mode) on or off and indicate the conditions under which you want memory allocations to fail. For example, you can cause memory allocation to fail for all tasks or for a specific task, for allocations within a specified size range, or for allocations after a certain number have already been performed. There are also additional tags that let you specify failure intervals, output verbosity, and other options.

After you finish using any of these calls, turn off memory debugging before you create the final version of your program. Enabling memory debugging incurs an overhead of 32 bytes per allocation made. If you use the `MEMDEBUGF_PAD_COOKIES` option, this overhead grows to 64 bytes per allocation.

Example: Allocating and Deallocating Memory

Most memory allocation involves nothing more than calling `AllocMem()` to allocate memory and `FreeMem()` to free memory. Example 5-1 demonstrates memory allocation and deallocation in an M2 program.

Example 5-1 *Allocating memory.*

```
#include <kernel/types.h>
#include <kernel/mem.h>
#include <kernel/stdio.h>

int main(int32 argc, char **argv)
{
    void *memBlock;

    memBlock = AllocMem(123, MEMTYPE_NORMAL);
        /* allocate any type of memory */
    if (memBlock)
    {
        /*
         * memBlock now contains the address of a block
         * of memory 123 bytes in size
         */

        FreeMem(memBlock, 123);          /* return the memory */
    }
}
```

```
    }  
    else  
    {  
        printf("Could not allocate memory!\n");  
    }  
}
```

Other Memory Topics (Caches, Bit Arrays)

Caches

Caches are a vital part of the M2 architecture. They allow the CPU to achieve its high throughput. Many critical system operations, such as DMA-based I/O, require very careful attention so as not to encounter coherency problems with respect to the caches.

Portfolio normally manages caches for you transparently, so your programs almost never have to be concerned with caches. There may be times, however, when you can increase performance by exercising some degree of control over caches. In even rarer cases, you may have to perform some cache management for the system to work at all. In case you do encounter these kinds of situations, the kernel offers a limited number of cache management calls.

Caches are inherently tied to a particular implementation of a particular CPU. So it is important to abstract details about the cache as much as possible in order to ensure forward compatibility with new processors. Because caches are CPU-specific, you should never assume anything about the size of the cache, the number of cache lines, etc. You can call the `GetCacheInfo()` function to obtain this information dynamically:

```
void GetCacheInfo(CacheInfo *info, uint32 infoSize);
```

The first argument to the `GetCacheInfo()` function is a pointer to a `CacheInfo` structure that is to receive the cache information being requested. The second argument is always set to the size of the `CacheInfo` structure, namely `sizeof(CacheInfo)`.

The fields of the `CacheInfo` structure are:

- ◆ **cinfo_Flags**. Provides a number of flags that show the state of the caches. The following flags are currently defined:

CACHE_UNIFIED. If this flag is set, it means there is only a single unified cache instead of Harvard-style split instruction and data caches.

CACHE_INSTR_ENABLED. If this flag is set, it means the instruction cache is on.

CACHE_INSTR_LOCKED. If this flag is set, it means the instruction cache is locked.

CACHE_DATA_ENABLED. If this flag is set, it means the data cache is on.

CACHE_DATA_LOCKED. If this flag is set, it means the data cache is locked.

- ◆ **cinfo_ICacheSize** and **cinfo_DCacheSize.** Specifies the number of bytes in the primary CPU instruction and data caches.
- ◆ **cinfo_ICacheLineSize** and **cinfo_DCacheLineSize.** Specifies the number of bytes in one line of the primary CPU instruction and data cache.
- ◆ **cinfo_ICacheSetSize** and **cinfo_DCacheSetSize.** Specifies the level of set associativity implemented in the primary CPU instruction and data cache.

As you can see, you can obtain a considerable amount of information about caches by calling the `GetCacheInfo()` function. Another thing you can do with caches is to cause the contents of the data cache to be written to memory. Writing cached data to memory can be quite expensive in terms of performance, however, so you should keep such operations to a minimum in order to keep CPU throughput high.

To help make the writing of cached data as efficient as possible, Portfolio provides a simple mechanism for keeping the need to flush the caches to a minimum. This mechanism is the `GetDCacheFlushCount()` function, which returns the current system flush count for the data cache:

```
uint32 GetDCacheFlushCount(void);
```

The value returned by `GetDCacheFlushCount()` is a count of the number of times the data cache has been flushed. When you want to write back the contents of the data cache to memory, the various functions that you must call to issue such a request expect you to supply a cache flush count. If the flush count you supply is different than the current system flush count, these functions don't do anything but return.

To illustrate the benefits of using `GetDCacheFlushCount()`, imagine the some data is being prepared in memory for a DMA engine to process. Before the DMA engine does its work, it is important to write back the contents of the caches to memory so that the DMA device can see the data it is supposed to. Once the data is prepared and ready to go, the task gets the current system flush count and waits for the DMA device to become available. Once the device is available, the task tries to flush the caches to memory. The flush is performed only if the system flush count matches the supplied flush count. If the two counts don't match, it

means that something else in the system has flushed the caches. And that means, in turn, that flushing the caches again would be redundant and would waste CPU time.

The available cache calls are:

- ◆ `void FlushDCacheAll(uint32 flushCount);`

This call writes back any modified data in the data cache to memory, and invalidates the cache.

- ◆ `void WriteBackDCache(uint32 flushCount, const void *start, uint32 numBytes);`

If the cache contains any modified data within the memory range supplied, this data is written back to memory. The contents of the cache remains valid after this operation.

- ◆ `void FlushDCache(uint32 flushCount, const void *start, uint32 numBytes);`

If the cache contains any modified data within the memory range supplied, this data is written back to memory, and the affected cache lines are invalidated.

Bit Arrays

The kernel provides a number of functions to deal efficiently with arrays of bits, or *bit arrays*. A bit array is a table of 32-bit words (uint32). Each bit in this table is addressable.

The functions discussed in this section can be used for working with bit arrays as follows:

- ◆ Setting and Clearing Bits
- ◆ Testing Bits
- ◆ Searching For Bits
- ◆ Atomically Setting and Clearing Bits
- ◆ Debugging Bit Arrays

Setting and Clearing Bits

You can set and clear bits within a bit array by using `SetBitRange()` and `ClearBitRange()`:

```
void SetBitRange(uint32 *array, uint32 firstBit, uint32 lastBit)
```

```
void ClearBitRange(uint32 *array, uint32 firstBit, uint32 lastBit)
```

You give both functions a pointer to the array of bits, and the number of the first bit in the range and the last bit in the range. All bits within the range are then set or cleared accordingly

Testing Bits

You can test whether ranges of bits within a bit array are currently all set to 1 or all set to 0, or if individual bits within the array are set to 0 or 1.

```
bool IsBitRangeClear(const uint32 *array, uint32 firstBit, uint32
    lastBit)

bool IsBitRangeSet(const uint32 *array, uint32 firstBit, uint32
    lastBit)

bool IsBitSet(const uint32 *array, uint32 bit)

bool IsBitClear(const uint32 *array, uint32 bit)
```

You must supply a pointer to the array of bits, and a range of bits to test, or a single bit to test.

Searching For Bits

You can search an array of bits looking for a range of bits that are all clear or all set. The search is done using a modified Boyer-Moore algorithm, so it is quite efficient. The longer the range of bits being sought, the faster the routines can go.

```
int32 FindSetBitRange(const uint32 *array, uint32 firstBit, uint32
    lastBit, uint32 numBits)

int32 FindClearBitRange(const uint32 *array, uint32 firstBit, uint32
    lastBit, uint32 numBits)
```

You give both functions an array of bits, a range of bits to limit the search to, and the number of contiguous set or clear bits you are looking for. The functions return the bit number of the beginning of the range, or a negative number if no suitable bit range was found.

Atomically Setting and Clearing Bits

In a multitasking environment, it is often useful to access memory locations in an atomic manner. That is, in a way that is protected from interference from other tasks running in the system.

The PowerPC documentation supplies a number of examples showing how to do a variety of atomic operations. The bit array package includes the following two routines that go along the same lines:

```
uint32 AtomicSetBits(uint32 *addr, uint32 bits)

uint32 AtomicClearBits(uint32 *addr, uint32 bits)
```

These routines set or clear specific bits within a particular word of memory. The access is guaranteed to be atomic, so it is safe to use these calls to coordinate between multiple tasks. This lets you implement your own safe and very low overhead semaphores.

The value returned by the functions corresponds to the value that was stored at the given memory location after the bits were set or cleared.

Debugging Bit Arrays

The `DumpBitRange()` function lets you dump an array of bits to the Debugger Terminal window in an easy-to-read format. This procedure can be useful when you want to make a quick determination of the current state of a bit array:

```
void DumpBitRange(const uint32 *array, uint32 firstBit, uint32  
    lastBit, const char *banner)
```

You give this function a pointer to an array of bits, a range of bits to display, and an optional banner line that is printed before the array itself, which is used to identify the particular dump sequence.

Managing Items

This chapter provides the programming details necessary to create or open items, manage items and, when finished, delete or close items.

This chapter contains the following topics:

Topic	Page Number
About Items	67
Creating an Item	69
Using an Item	73
Deleting an Item	75
Opening an Item	75
Closing an Item	75

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About Items

Items are system-maintained handles for shared resources. Each item contains a globally unique identification number and a pointer to the resource. Items can refer to any one of many different system components such as data structures, I/O devices, folios, tasks, and so on. You need to understand and use the item function calls in order to use these various system components.

The procedure for working with an item is the same, regardless of the item type. The procedure is as follows:

1. Define the necessary parameters (tag arguments) for the item you want.
2. Create or open the item using the tag arguments.
3. Use the item.
4. Delete or close the item when you're finished with it.

Portfolio provides a set of kernel calls that manage items while they're in use. These calls allow a task to check for the existence of an item, find the item number of a named object, get a pointer to an item, and change the item's priority and owner.

The kernel creates and keeps track of items in system memory. When the kernel creates an item, it creates a node for that item using the `ItemNode` data structure:

Example 6-1 `ItemNode` structure

```
typedef struct ItemNode
{
    struct ItemNode *n_Next;           /*pointer to next itemnode in list*/
    struct ItemNode *n_Prev;           /*pointer to previous itemnode in list*/
    uint8 n_SubsysType;                /* what folio manages this node */
    uint8 n_Type;                      /* what type of node for the folio */
    uint8 n_Priority;                  /* queueing priority */
    uint8 n_Flags;                     /* misc flags, see below */
    int32 n_Size;                      /* total size of node including hdr */
    char *n_Name;                      /* ptr to null terminated string or NULL*/
    uint8 n_Version;                   /* version of of this itemnode */
    uint8 n_Revision;                  /* revision of this itemnode */
    uint8 n_Reserved0;                 /* Reserved */
    uint8 n_ItemFlags;                 /* additional system item flags */
    Item n_Item;                       /* ItemNumber for this data structure */
    Item n_Owner;                      /* creator, present owner, disposer */
    void *n_Reserved1;                 /* Reserved pointer */
} ItemNode, *ItemNodeP;
```

The fields in this data structure define the status of the item and give information about the item. User tasks can't change their values, but they can look at them for information such as the version and revision numbers of an item (useful for items such as folios or devices). `LookupItem()`, described later, returns a pointer to the `ItemNode` structure for a specific item.

Creating or Opening an Item

To use most items such as message ports and semaphores, a task must first create them. To use a folio or a device, a task opens it instead of creating it from scratch.

The difference between item creation and item opening is important: a created item is owned by the creating task, which can delete the item at any time. A created item is automatically deleted if the owning task quits without deleting it. On the other hand, an opened item is owned by the original creator and is shared among tasks. When a task opens the item, the item is loaded into memory and the kernel registers the opening task as a user. When another task opens the same item, the kernel registers that task as well.

Creating an Item

To create an item, use the `CreateItem()` function call:

```
Item CreateItem( int32 ctype, const TagArg *p )
```

`CreateItem()` takes two arguments. The first, `ctype`, specifies the type of folio in which the item is used and, the type of item within that folio. This value must always be present—it defines the item type. To generate this value, use the `MkNodeID()` macro (described later), which accepts the folio type and item type values, and returns the proper value for `CreateItem()`.

The second argument, `p`, is a pointer to additional arguments that define the characteristics of the item. These arguments, known as tag arguments, are specific to each item type, and are discussed later in the chapter.

`CreateItem()` returns the item number of the newly created item or an error code (a negative value) if an error occurs.

Specifying the Item Type

To come up with the item type value for `CreateItem()`, use `MkNodeID()`:

```
int32 MkNodeID( int32 folioNum, int32 itemTypeNum )
```

The macro accepts two values: `folioNum`, which specifies the item number of the folio responsible for the item, and `itemTypeNum`, which specifies the item type number for the item being created. This item type number is folio-specific. Both values are set by constants defined within the header file for the appropriate folio. For example, `<audio/audio.h>` defines `AUDIONODE` as the constant that specifies the Audio folio, and lists audio folio item constants such as `AUDIO_TEMPLATE_NODE`.

See the chapter dealing with Portfolio items in the *3DO M2 Portfolio Programmer's Reference* for a list of the constants specifying the item numbers for the folios and for the items in the folios. To use those constants, you must be sure to include the ".h" file of the proper folio. The constants for kernel items, such as message ports or tasks, are in the *<kernel/kernelnodes.h>* file.

Tag Arguments

The values of the tag arguments determine the values of the fields in an item's *ItemNode* structure. *CreateItem()* creates an *ItemNode* structure for a new item. The call reads the values of the tag arguments, interprets them, and writes the corresponding values in the *ItemNode* fields.

All item types use a standard set of tag arguments defined in *<kernel/item.h>*. Many items also use additional tag arguments to specify properties that are unique to those item types.

To define values for an item's associated tag arguments, you must first declare an array of type *TagArg*, which is defined in *<kernel/types.h>* as follows:

Example 6-2 *Defining values for an item's associated tag argument.*

```
typedef struct TagArg
{
    uint32  ta_Tag;
    TagData ta_Arg;
} TagArg, *TagArgP;
```

The first field, *ta_Tag*, is a value that specifies the type of tag argument. The second field, *ta_Arg*, is a 32-bit value that contains the actual argument.

The standard tag argument types for all items are defined in *<kernel/items.h>*. They include:

- ◆ **TAG_ITEM_PRI**. Sets the priority of an item.
- ◆ **TAG_ITEM_NAME**. Sets the name of an item.
- ◆ **TAG_ITEM_VERSION**. Sets the version number of an item.
- ◆ **TAG_ITEM_REVISION**. Sets the revision number of an item.
- ◆ **TAG_ITEM_UNIQUE_NAME**. Specifies the item must be uniquely named within its type.
- ◆ **TAG_ITEM_END** or **TAG_END**. Ends a list of tag arguments.

The value for each of these arguments is precisely the same as the value for corresponding fields in `ItemNode`. When an item is created, these values are simply written into the appropriate `ItemNode` fields.

The chapter dealing with Portfolio items in the *3DO M2 Portfolio Programmer's Reference* describes the custom tag arguments used for each item type.

When you provide tag arguments for an item, some of the arguments for that item are required and some are optional. You must provide values for required tag arguments. You must also provide specific values for optional tag arguments, because any values that you do not provide are filled in with default values by the operating system when it creates the item. Defaults aren't guaranteed to be 0, so you must provide the value of 0 if that's what you want set for a tag argument.

If an entire array of tag arguments is optional and you want to use the system default values for all the arguments, then you do not need to declare the `TagArg` array in your application. Instead, pass `NULL` as the pointer to the tag argument. When you create a tag argument array, its general format is:

Example 6-3 *Tag arguments array.*

```
TagArg          TagArrayName[] =
{
    command tag,value assigned,
    ... ..
    TAG_END, 0
};
```

`TagArrayName` is the name of the array for the tag argument values; `command tag` is the name of the tag argument, and `value assigned` is the value assigned to that tag argument. The last tag argument in an array is always called `TAG_END`, and must be set to 0. This signals the end of the tag argument list.

VarArgs Tag Arguments

Most functions in Portfolio that accept tag arguments have a split personality: The standard version of a function has a parameter of type `const TagArg *`, the `VarArgs` version of a function, is denoted with the `VA` suffix (`CreateItemVA()` for example). The `VarArgs` functions offer a more convenient way to provide the system with tag arguments.

When you use the `VarArgs` version of a function, you do not need to create arrays of `TagArg` structures to pass to the function. Instead, you can construct a tag list on the stack directly within the function call. Example 6-4 shows a standard call with `TagArg` structures:

Example 6-4 *A standard function call with TagArg structures.*

```
{
TagArg tags[3];

    tags[0].ta_Tag = CREATEMSG_TAG_REPLYPORT;
    tags[0].ta_Arg = (void *)replyPort;
    tags[1].ta_Tag = CREATEMSG_TAG_MSG_IS_SMALL;
    tags[1].ta_Arg = 0;
    tags[2].ta_Tag = TAG_END;

    item = CreateItem(MKNODEID(KERNELNODE, MESSAGENODE), tags);
}
```

Instead of this cumbersome and error prone array, you can build the tag list on the stack using the `CreateItemVA()` function as shown in Example 6-5:

Example 6-5 *Using VarArgs to provide tag arguments.*

```
{
    item = CreateItemVA(MKNODEID(KERNELNODE, MESSAGENODE),
        CREATEMSG_TAG_REPLYPORT,    replyPort,
        CREATEMSG_TAG_MSG_IS_SMALL, 0,
        TAG_END);
}
```

This method of providing tag arguments is much easier to read and maintain. There is no casting needed, no array to maintain, and so on.

The Item Number

Once an item is created, the call that creates it returns a positive 32-bit number, which is the unique item number for the new item. A task uses this number whenever it needs to refer to the item in later calls—this item number is important, and should be stashed away in a handy variable. If an error occurs, a negative number is returned.

Convenience Calls to Create Items

Now that you've seen the details of creating items using `CreateItem()`, you should know that most folios provide higher-level convenience calls that do the same thing for commonly used items in a folio, which makes programming much easier. The convenience calls usually take arguments that define the

characteristics of the item, translate arguments into the appropriate tag arguments, and then call a lower-level item creation call to create the item. Some of those calls include:

- ◆ `CreateIOReq()` creates an `IOReq`.
- ◆ `CreateMsgPort()` creates a message port.
- ◆ `CreateMsg()` creates a message.
- ◆ `CreateSemaphore()` creates a semaphore.
- ◆ `CreateThread()` creates a thread.

Generally, use a convenience call whenever one is available for the type of item you want to create. You'll find details about these calls in the *3DO System Programmer's Reference* and in the chapters of this volume that discuss specific folios.

Using an Item

Once an item is created or opened, you can manage it with the kernel item calls, which are described here.

Finding an Item

To find an item by specifying its item type and a set of tag arguments or a name, use these calls.

```
Item FindItem( int32 cType, const TagArg *tp )
Item FindItemVA( int32 cType, uint32 tags, ... )
Item FindNamedItem( int32 cType, const char *name )
```

`FindItem()` accepts an item type value, which is created by `MkNodeID()` as it is for the `CreateItem()` call. This value specifies the type of the item for which the call should search. The second argument, `tp`, is a pointer to an array of tag arguments the call should try to match. When the call is executed, it looks through the list of items and, if it finds an item of the specified item type whose tag arguments match those set in the tag argument array, it returns the item number of that item.

`FindNamedItem()` calls `FindItem()` after creating a `TagArg` for the name. When it searches through the item list, it checks names of all items of the specified type. If it finds a match, it returns the item number of that item.

All of the item finding calls return a negative number if an error occurred in the search.

Getting a Pointer to an Item

To find the absolute address of an item, use this call:

```
void *LookupItem( Item i )
```

`LookupItem()` takes the item number of the item to locate as its only argument. If it finds the item, it returns a pointer to the item's `ItemNode` structure; if it doesn't find the item, it returns `NULL`. Once you have a pointer to the item, you can use it to read the values in the fields of the data structure. The `ItemNode` structure is protected by the system, so a user task can't write to it.

Checking If an Item Exists

To see if an item exists, use this call:

```
void *CheckItem( Item i, uint8 ftype, uint8 ntype )
```

`CheckItem()` accepts the item number of the item you want to check, followed by the item number of the folio and the item type number of the item. (`ftype` and `ntype` are the same as the arguments accepted by `MkNodeID()`, and are supplied with constants defined in appropriate header files.) When executed, the call returns a `NULL` if it can't find an item with the specified item number, or if it found an item but it didn't have the specified folio and node types. If the call finds an item that matches in all respects, it returns a pointer to that item.

Changing Item Ownership

To change the ownership of an item from one task to another, use this call:

```
Err SetItemOwner( Item i, Item newOwner )
```

The first argument of `SetItemOwner()` is the item number of the item for which to change ownership; its second argument is the item number of the task that is to be the new owner. The task that makes this call must be the owner of the item. If ownership transfer fails, the call returns a negative number (an error code).

When ownership of an item changes, the item is removed from the former owner's resource table and placed in the new owner's resource table. This allows an item to remain in the system after the original owner dies; all items owned by a task are removed from the system when the task dies.

Changing an Item's Priority

To change the priority of an item, a task should use this call:

```
int32 SetItemPri( Item i, uint8 newpri )
```

`SetItemPri()` accepts the item number of the item whose priority you want to change, followed by an unsigned 8-bit integer which specifies the new priority for the item. Priority values range from 0–255. The call returns the former priority value of the item if successful, otherwise it returns a negative number.

Deleting an Item

When a task finishes with an item it owns, it should delete the item from the system to free resources for other uses. The task can use this call:

```
Err DeleteItem( Item i )
```

`DeleteItem()` accepts the item number of the item to delete. It returns 0 if successful, or an error code (a negative value) if an error occurs.

Only use `DeleteItem()` to delete items created with `CreateItem()`. If you've used a convenience call to create an item, you must use the corresponding deletion routine. A task must own an item to delete it.

Once an item is deleted, the kernel does not reuse the item number when it creates new items, to maintain the integrity of items. When a task tries to use an item number for a deleted item, the kernel informs the task that the item no longer exists.

Opening an Item

When a task uses a system-supplied item such as a device or a folio, it opens the item instead of creating it. To open an item, use these calls:

```
Item OpenItem( Item FoundItem, void *args )
```

```
Item FindAndOpenItem( int32 cType, const TagArg *tags)
```

```
Item FindAndOpenItemVA( int32 cType, uint32 tags, ...)
```

`OpenItem()` accepts the item number of the resource to be opened (which is found with `FindItem()`) and a pointer. Currently, NULL should always be passed in for this argument.

`FindAndOpenItem()` finds an item and opens it with one call. It works just like `FindItem()`, except that it automatically opens the item before returning it to you.

Closing an Item

When a task finishes using an opened item, it should close it so the kernel knows the item is no longer required by that task. When no tasks require an opened item, the kernel can remove the item from memory, freeing system resources.

To close an item, use this call:

```
Err CloseItem( Item i )
```

`CloseItem()` accepts the item number of the item to close. It returns 0 if successful or an error code (a negative value) if an error occurs.

Semaphores

This chapter explains techniques for sharing system resources using semaphores.

This chapter contains the following topics:

Topic	Page Number
About Semaphores	77
Creating a Semaphore	79
Locking a Semaphore	79
Exclusive and Shared Access	80
Priority Inversion	80
Unlocking a Semaphore	80
Finding a Semaphore	81
Deleting a Semaphore	81

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About Semaphores

Tasks use special items known as *semaphores* to control access to shared resources. The following sections explain how they work.

The Problem: Sharing Resources Safely

A classic problem in computing is allowing multiple tasks to share the same resource. The problem affects any resource, including memory or a disk file, whose state can be changed by the tasks that share it. If a task changes a resource before another task finishes using it, and the change violates an assumption the first task has about the resource's contents, crashes and corruption can occur.

The Solution: Locking Shared Resources When Using Them

To share a resource safely, a task must be able to lock the resource while it's using it and unlock it when it finishes using it. This is known as mutual exclusion: Other tasks are excluded from the resource until it is safe for them to use it.

The Implementation: Semaphores

There are many techniques for enforcing mutual exclusion. One of the simplest and most effective techniques uses special data structures known as semaphores. A semaphore keeps track of how many tasks are using or waiting to use a shared resource. This approach—described in many good textbooks on operating systems—is the one Portfolio uses. Tasks using Portfolio semaphores use the functions provided by the kernel for creating semaphores, for deleting them, and for locking and unlocking the resources they control.

On 3DO systems, it is up to each task to ensure that any shared resources it owns—in particular, memory pages that other tasks can write to—are shared safely. (To learn how tasks can grant other tasks write access to their memory, see Chapter 5, “Managing Memory”). Although tasks are not required to use semaphores, a program should always use them, or another mutual-exclusion mechanism, when a task shares any of its memory with another task.

Using Semaphores

To control access to a shared resource it owns, a task

- ◆ Creates a semaphore
- ◆ Associates the semaphore with the resource
- ◆ Tells the other tasks that share the resource there is a semaphore for controlling access to the resource

Each task then must lock the semaphore before using the resource by calling `LockSemaphore()` and unlocks it as soon as it's done using it by calling `UnlockSemaphore()`. Except for creating the semaphore by calling `CreateSemaphore()`, sharing a resource requires the cooperation of the tasks that are involved. The tasks must agree on a way to communicate the information

about the new semaphore, such as intertask messages or shared memory, and they need to agree on a format for this information. They must also agree to lock the semaphore before using the resource and unlock it when they're done.

If your task needs access to a shared resource and cannot continue without it, you can ask the kernel to put the task into wait state until the resource becomes available. You do this by including the `SEM_WAIT` flag in the `flags` argument of `LockSemaphore()` discussed below.

Creating a Semaphore

To create a semaphore, use `CreateSemaphore()`:

```
Item CreateSemaphore( const char *name, uint8 pri )
```

The `name` argument is the name to give to the new semaphore. You can use the `FindSemaphore()` macro described later in this chapter to find the semaphore by name. To ensure that `FindSemaphore()` finds the correct semaphore, you must provide a unique name for each semaphore you create.

The `pri` argument determines where this semaphore will be placed within the list of all semaphores in the system. A higher priority puts it closer to the start of the list. `CreateSemaphore()` returns the item number of the new semaphore or an error code (a negative value) if an error occurred.

To delete a semaphore created with `CreateSemaphore()`, use the `DeleteSemaphore()` function described in "Deleting a Semaphore" on page 81.

Locking a Semaphore

To lock a semaphore, thereby preventing other tasks from accessing the associated resource until your task is finished with it, call the `LockSemaphore()` function:

```
int32 LockSemaphore( Item s, uint32 flags )
```

The `s` argument is the item number of the semaphore for the resource you want to lock. Use the `flags` argument to specify what to do if the resource is already locked: if you include the `SEM_WAIT` flag, your task is put into wait state until the resource is available.

`LockSemaphore()` returns 1 if the resource was locked successfully. If the resource is already locked and you did not include the `SEM_WAIT` flag in the `flags` argument, the function returns 0. If an error occurs, the function returns a negative error code.

Exclusive and Shared Access

It is possible to lock semaphores in one of two modes: *exclusive* or *shared*. Locking in exclusive mode means that only one task is allowed to have the semaphore locked at any one moment. Shared access allows multiple tasks that don't intend to modify the data to have the semaphore locked at the same time. This improves overall system performance by avoiding needless blocking on a shared resource.

When a task only needs to perform a read operation on a resource, it should lock the semaphore in shared mode. This allows other tasks that also merely want to read the resource to lock the semaphore at the same time. If a task intends on modifying the resource, it must lock the semaphore in exclusive mode.

When a semaphore is locked in shared mode, a new task that attempts to lock the semaphore in shared mode will be granted access. However, if the new task attempts to lock the semaphore in exclusive mode, it will be held off until all current shared lockers unlock the semaphore.

If a semaphore is locked in exclusive mode, or if there are any tasks waiting to lock the semaphore in exclusive mode, new tasks attempting to lock the semaphore will always be put at the end of the wait queue for the semaphore.

By default, `LockSemaphore()` locks semaphores in exclusive mode. To lock a semaphore in shared mode, you must supply the `SEM_SHAREDREAD` flag to the `LockSemaphore()` call.

Priority Inversion

Priority inversion is a common problem associated with mutual exclusion. Inversion occurs when a high-priority task is blocked waiting for a resource that a low-priority task owns. In effect, this means that the high-priority task is now running at the same priority as the lower-priority task, since it needs to wait for the lower priority task to unlock the resource until it can run again.

Portfolio currently doesn't provide any special mechanisms to cope with priority inversion. The situation occurs fairly infrequently, although it does happen. It is worth understanding the problem and working around it. A typical solution is to bump the priority of the lower priority task to match that of the highest priority task that is waiting on the resource.

Unlocking a Semaphore

To unlock a semaphore that you've locked, use the `UnlockSemaphore()` function:

```
Err UnlockSemaphore( Item s )
```

The `s` argument is the item number of the semaphore for the resource you want to unlock. The function returns 0 if successful or an error code (a negative value) if an error occurred. It returns the `NOTOWNER` error code if your task did not call `LockSemaphore()` for the specified semaphore.

Finding a Semaphore

To find a semaphore by name, use the `FindSemaphore()` macro:

```
Item FindSemaphore( const char *name )
```

The `name` argument is the name of the semaphore to find. The macro returns the item number of the semaphore or a negative error code if an error occurred.

Deleting a Semaphore

To delete a semaphore, use the `DeleteSemaphore()` function:

```
Err DeleteSemaphore( Item s )
```

The `s` argument is the item number of the semaphore to delete. The function returns zero or a positive integer if it is successful; otherwise, a negative error code if an error occurred. As with all items, the item number of a deleted semaphore is not reused.

Communicating Among Tasks

This chapter gives the background and necessary programming details to perform intertask communication.

This chapter contains the following topics:

Topic	Page Number
About Intertask Communication	83
Using Signals	84
Passing Messages	87

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About Intertask Communication

Two mechanisms can be used for intertask communication: signals and messages. Signals are used as an intertask flag with which one task can notify another that something has occurred. Messages not only allow one task to notify another, but also carry data from the sending task to the receiving task.

Signals and a task's ability to wait for a signal are the basis for all intertask communication. The message mechanism, which passes more detailed information, is built on top of the signal mechanism. Sending and receiving a message is based on a signal sent to the receiving task, informing it that a message is waiting.

Using Signals

The procedure for using a signal involves several steps:

- ◆ Task A uses `AllocSignal()` to allocate a signal bit (1 bit within a 32-bit word) for a signal.
- ◆ Task A communicates the allocated signal bit to Task B so that it knows which signal to send (using the allocated signal bit) back to Task A.
- ◆ Task A uses `WaitSignal()` to enter wait state until it receives a signal on the bit it allocated.
- ◆ Task B uses `SendSignal()` to send a signal to Task A. Task B must provide `SendSignal()` with the item of the task to signal, and the signals to send the task.
- ◆ Task A receives the signal and returns from wait state.
- ◆ Task A uses `FreeSignal()` to free the signal bit when it's no longer needed.

Allocating a Signal Bit

To allocate a signal bit for a task, use this call:

```
int32 AllocSignal( int32 sigMask )
```

`AllocSignal()` accepts as its sole argument a 32-bit word (`sigMask`) which specifies which signal bits to allocate. There are 31 signal bits for each task. The lower-8 bits (bits 0–7) are reserved for system use and cannot be allocated. Bits 8–30 can be allocated for task use. To specify a particular signal bit to allocate, just set the corresponding bit in the `sigMask` parameter. For example, to allocate bit 21, you set `sigMask` to `0x00200000`. If you don't care which signal bit you get, you can use `AllocSignal(0)`, in which case the kernel chooses a free bit and allocates it for you. It is most common to simply use `AllocSignal(0)` instead of requesting specific signal bits.

`AllocSignal()` returns a signal mask with bits set to 1 where signals were successfully allocated. If no bits were available for allocation, the call returns 0. If there was an error in allocation, the call returns an error code (a negative number).

The code fragment in Example 8-1 demonstrates using `AllocSignal()` to allocate a signal bit. In this program, a parent task creates two threads, and it uses the signal mechanisms for the threads to communicate with the parent task. Here, the parent task allocates two of its signal bits, one for each thread. First, it declares two global variables, `threadSig1` and `threadSig2`, to use as signal mask variables for each of the threads:

Example 8-1 *Allocating a signal bit.*

```
/* Global variables shared by all threads. */
static int32  thread1Sig;
static int32  thread2Sig;
...
...

/* allocate one signal bit for each thread */
thread1Sig = AllocSignal(0);
thread2Sig = AllocSignal(0);

if ((thread1Sig == 0) || (thread2Sig == 0))
{
    /* could not allocate the needed signal bits */
}
```

The two calls to `AllocSignal()` allocate 2 available signal bits. If successful, the variables `threadSig1` and `threadSig2` each contain a signal mask with a specially allocated bit for each thread to use.

Receiving a Signal

To receive a signal, a task enters wait state using this call:

```
int32 WaitSignal( uint32 sigMask )
```

`WaitSignal()` accepts a signal mask that specifies which of the task's allocated signal bits to wait for. It can specify more than one bit to wait for, but it can't specify any bits that haven't already been allocated. If unallocated bits are specified, the call returns a negative value (an error code) when it executes. Before returning, `WaitSignal()` clears the signal bits.

When the allocated bits are specified and the call executes, the task enters wait state until a signal is sent to one of the signal bits it specified in `WaitSignal()`. The task then exits wait state and returns to the ready queue. The call returns a signal mask with 1 set in each bit where a signal was received.

A task can receive signals before it enters wait state, in which case they're kept as pending signals in a pending signals mask maintained in the task's control block (TCB). There is no signal queue—that is, no more than one signal can be kept pending for each allocated signal bit. When a task calls `WaitSignal()`, any pending signals that match the signal mask given to `WaitSignal()` cause the call to return immediately, and the task never enters wait state. The signal bits that the task was waiting for are cleared before returning.

Sampling and Changing the Current Signal Bits

It is sometimes necessary for a task to look at the current state of the signal bits to determine whether a given signal was received. To do this, use

`GetCurrentSignals()`:

```
int32 GetCurrentSignals(void)
```

The macro returns a signal mask with bits on for every signal bit that has been received. This provides a sample of the current signal bits without entering wait state.

You can forcibly set the signal bits of your task by using `SendSignal()` with 0 for the task item. This is sometimes useful to set the initial state of the task's signals.

You can also forcibly clear signal bits of a task by using the macro `ClearCurrentSignals()`.

```
Err ClearCurrentSignals(int32 signalMask)
```

This macro takes a signal mask that describes the signal bits to clear. The macro returns a negative value if reserved signal bits (bits 0–7) are specified.

Sending a Signal

A task sends one or more signals to another task with this call:

```
Err SendSignal( Item task, int32 sigMask )
```

The call accepts the item number of the task to which the signal or signals should be sent, and a signal mask with a 1 in each bit to which a signal is sent. When it executes, the kernel checks that each specified signal bit in `SendSignal()` is allocated by the receiving task. If it's not, the call returns a negative number (an error code).

If all specified bits are allocated by the receiving task, the kernel performs a logical OR between the `SendSignal()` signal mask and the pending signals mask maintained in the receiving task's TCB. This is how the pending signals mask maintains a record of pending signals. When the receiving task uses `WaitSignal()`, the kernel checks the pending signals mask, and if any bits are set to 1, the task immediately returns to execution.

Freeing Signal Bits

When a task no longer needs an allocated signal bit, it uses this call:

```
Err FreeSignal( int32 sigMask )
```

The call accepts a signal mask with a 1 set in each bit that is to be freed, and a 0 set in each bit where allocation status should be unchanged. For example, if bits 8 and 9 are currently allocated and you wish to free only bit 8, then you need to create a signal mask with the binary value $1 \ll 8$ (which equals a 256 decimal value).

`FreeSignal()`, frees the specified bits when executed. It returns 0 if it was successful, and a negative number (an error code) if unsuccessful.

Sample Code for Signals

For the full code sample on which the previous code sample is based, see Example 2-2 on page 28, which is also an example for using threads. The variables declared at the beginning of the example are global to the `main()` function (the parent task) and the two threads. These global variables pass signal masks between the threads and the task. The `main()` function, which appears at the end of the example, allocates signals for the two threads it will create.

The `main()` function uses `WaitSignal()` to wait for signals from the two threads. The threads use `SendSignal()` to send their signals to the parent task.

Passing Messages

The message system allows tasks to send data to one another, providing information flow from task to task. Tasks must create the following elements for the message system to work:

- ◆ The sending task must create a message.
- ◆ The receiving task must create a message port.

Once the elements are in place, the sending task follows these procedures:

- ◆ It specifies a message to send and a destination message port to which to send the message.
- ◆ It specifies the data to go with the message.

The receiving task does one of the following procedures:

- ◆ It enters wait state to wait for a message on a specified message port.
- ◆ It checks one of its message ports to see if a message has arrived there.

Once a message arrives at a task's message port, the task follows these procedures:

- ◆ It removes the message from its message port.
- ◆ It gets a pointer to the message, and reads data directly from the message, or finds the data block pointer and size and reads from the data block.

- ◆ If the message needs a reply, the task writes new data directly into the message or sets a pointer and size in the message for a new data block, and returns the message to its owner.

The message system is very flexible, allowing a single task to create multiple message ports and multiple messages. A single message port can receive many messages; they're queued up and retrieved one at a time by the receiving task. A message port allows a single task to receive, in serial order, messages from many other tasks.

Messages, like all other items, are assigned priorities. Message priority determines the ordering of messages within a message queue: higher-priority messages go to the front of the queue. Messages with the same priority are arranged in arrival order: earlier messages come first.

Creating a Message Port

Before a task can receive a message, it must have created at least one message port. To create a message port, use this call:

```
Item CreateMsgPort( const char *name, uint8 pri, int32 signal )
```

The call accepts three arguments: a name string, *name*, which it uses to name the message port; a 1-byte priority value, *pri*, which assigns a priority to the message port; and a 32-bit signal mask, *signal*, which assigns an allocated signal bit to the message port.

The message port's name and priority don't change the way the message port operates, but are useful when another task tries to find the message port. Ports are kept in a list which is sorted by priority. When searching for a message port by name, the kernel finds the port of that name with the highest priority before other ports of the same name.

The signal mask specifies a signal to associate with the message port. The signal mask must contain only allocated signal bits. If you specify a signal mask of 0, `CreateMsgPort()` allocates a signal bit automatically. When the port is later deleted using `DeleteMsgPort()`, the signal bit is automatically freed.

`CreateMsgPort()` returns the item number of the newly created message port if successful, or returns a negative number (an error code) if unsuccessful. The message port now exists as an item in system memory, owned by the creating task.

The message port uses its assigned signal bit to signal its owner task whenever a message arrives at the port. The signal bit should *not* be freed until the message port is freed.

The item number returned by `CreateMsgPort()` is a handle to the new port. The owner task should give it out to any task that may want to send a message to it. Sending a message requires specifying the item number of the message port to which the message is sent.

Creating a Message

To create a message, a task must specify which of three message types to create:

- ◆ A **standard message**, which has a pointer to a block of data and the size of that data.
- ◆ A **small message**, which has two 32-bit words of data.
- ◆ A **buffered message**, which has a built-in data block.

If the task sends eight or fewer bytes of data within the message itself, it should create a small message.

If the task sends more than eight bytes of data and only points to the data in a data block that isn't carried with the message, it should create a standard message.

If the task sends more than eight bytes of data *within* the message, it must create a buffered message.

Creating Standard Messages

To create a standard message, use this call:

```
Item CreateMsg( const char *name, uint8 pri, Item mp )
```

The first argument, `name`, is the name for the message, which is useful for finding the message later. The second argument, `pri`, sets the message's priority, which determines how it is sorted in a receiving task's message queue. It can be a value from 0 to 255. The third argument, `mp`, is the item number of a message port that belongs to the creating task. This is the message's optional reply port, where it returns if the receiving task sends it back with a reply call. Supply 0 for this last argument if the message doesn't have a reply port.

A standard message carries with it a pointer to a data block, and the size of the data block. The task that receives the message can obtain the pointer to the data block and access the data. If the receiving task has write permission to the pages of memory where the data block is located, it can even write to the data block directly. Because you pass a pointer to a data block, standard messages are often referred to as "pass-by-reference" messages.

The reply to a standard message can contain four bytes of data in the `msg_Result` field of the `Message` structure.

When executed, `CreateMsg()` creates a standard message and returns the message's item number. If unsuccessful, it returns a negative number (an error code).

Creating Buffered Messages

Some applications require a message that can carry data within the message itself. This lets the sender copy data into the message, send the message, and not have to keep around its original data buffer. The buffer is no longer needed because the data is now in the message itself. In addition, when the destination task returns a buffered message, it can also write data to the message's buffer, allowing a great deal of information to be returned.

Because buffered messages require data to be copied into them before being sent, they are often referred to as "pass-by-value" messages.

With buffered messages, a sending task can send large amounts of read/write data to another task. The receiving task can modify this data and return it to the caller. This avoids granting the receiving task write permission to memory buffers in the sending task's address space.

To create a buffered message, use this call:

```
Item CreateBufferedMsg( const char *name, uint8 pri, Item mp, uint32
                        datasize )
```

Like `CreateMsg()`, this call accepts arguments that point to a name, supply a priority, and optionally give the item number of a reply port for the message. The additional argument, `datasize`, specifies the size, in bytes, of the data buffer to create.

When the call executes, it creates a buffered message with its accompanying buffer. If successful, the call returns the item number of the message. If unsuccessful, it returns a negative number (an error code).

Creating Small Messages

Both standard and buffered messages have message structure fields with pointers to a data block and the size of the data block. A small message uses those same fields to store up to eight bytes of data instead of storing a pointer and a block size value.

To create a small message, use `CreateSmallMsg()`:

```
Item CreateSmallMsg( const char *name, uint8 pri, Item mp )
```

The arguments are identical to those for `CreateMsg()`. You'll see the difference when you call `SendSmallMsg()` (described later).

Sending a Message

A task can send all three types of messages, but each requires different handling.

Sending Small Messages

To send a small message, use this call:

```
Err SendSmallMsg( Item mp, Item msg, uint32 val1, uint32 val2 )
```

The first argument, `mp`, is the item number of the message port to which the message is sent. The second argument, `msg`, is the item number of the small message to send. The third argument, `val1`, is the first four bytes of data to send in the message; the fourth argument, `val2`, is the second four bytes of data to send in the message. (If you don't know the item number of the message port, but you know its name, you can use `FindMsgPort()` to get the item number.)

When `SendSmallMsg()` executes, it writes the first value into the `msg_Val1` field of the message's data structure and the second value into the `msg_Val2` field of the structure. It then sends the message to the specified message port. It returns a 0 if the message is successfully sent, or a negative number (an error code) if unsuccessful.

Sending Standard and Buffered Messages

To send a standard or buffered message, use this call:

```
Err SendMsg( Item mp, Item msg, const void *dataptr, int32 datasize )
```

Like `SendSmallMsg()`, this call accepts the item number of a message port to which to send the message, `mp`, and the item number of the message to send, `msg`. Instead of accepting values to store with the message, `SendMsg()` accepts a pointer to a data block in the `dataptr` argument and the size, in bytes, of the data block in the `datasize` argument.

When `SendMsg()` executes, it sends the message, returning a value of 0 or greater if successful, or a negative number (an error code) if unsuccessful. Its effect on the message depends on whether the message is standard or buffered.

A standard message stores the data block pointer and the data size value in the message. The data block remains where it is, and the receiving task reads it there, using the pointer and size value to find it and know how far it extends.

With a buffered message, `SendMsg()` checks the size of the data block to see if it will fit in the message's buffer. If it won't, the call returns an error. If it does fit, `SendMsg()` copies the contents of the data block into the message's buffer. The receiving task can then read the data directly out of the message's buffer.

The advantage of a buffered message is that it carries the full data block within its buffer, so the sending task can immediately use the original data block's memory for something else. The task doesn't need to maintain the data block until it's sure that the receiving task has read the block. The data block is handled by the system, and is carried within the message in protected system memory.

Receiving a Message

Once a message is sent to a message port, it is held in the message port's message queue until the receiving task retrieves the message, or the sending task decides to pull it back. The messages within the queue are put in order first by priority, and then by order received.

To receive a message, a task can wait until it receives notification of a message at a message port, or it can check a message port directly to see if a message is there.

Waiting for a Message

To enter wait state until a message arrives on a specific message port, a task uses this call:

```
Item WaitPort( Item mp, Item msg )
```

`WaitPort()` accepts the required argument, `mp`, which contains the item number of the message port where the task receives messages. The call also accepts an optional argument, `msg`, which contains the item number of a specific message the task expects to receive at the message port. (To wait for any incoming message, use 0 as the value of `msg`.)

Note: *If the message arrived before `WaitPort()` is called, the task never enters wait state.*

When `WaitPort()` executes, the task enters wait state until it detects a message on the message port's message queue. At that point, if the call doesn't specify a specific message, the task exits wait state, and the call returns the item number of the first message in the port's queue. The message is removed from the message queue, and the task can use its item number to find and read the message.

If a specific message was given to `WaitPort()` in addition to a message port, the task waits until that message arrives at the message port. When it arrives, the task exits wait state and the message is removed from the message queue.

Checking for a Message

To check for a message at a message port without entering wait state, a task uses this call:

```
Item GetMsg( Item mp )
```

`GetMsg()` accepts a single argument: the item number of the message port where it wants to check for messages. When it executes, it checks the message port to see if there are any messages in the port's queue. If there are, it returns the item number of the first message in the queue and removes it from the queue. If there is no message at the message port, the call returns 0. If there is an error, the call returns an error code (a negative number).

Working with a Message

Once a task receives a message, it interprets the contents of the message to determine what the message means. To do this, the task must obtain a pointer to the `Message` structure associated with the message's item number:

```
Message *msg;
Item      msgItem;

msg = (Message *)LookupItem(msgItem);
```

Once the task has the pointer, it can determine the type of message it has received by inspecting the contents of the `msg.n_Flags` field of the `Message` structure. If the `MESSAGE_SMALL` bit is set in this field, it means it is a small message. If the `MESSAGE_PASS_BY_VALUE` bit is set, it means it is a buffered message. If neither of these bits is set, it means that it is a standard message.

When you set up communication channels among your tasks or threads using message ports, you should know what type of messages will be sent around to these ports, so that you don't always need to check the types.

Working with Small Messages

When you receive a small message, the only information you can get from the message is contained in the `msg_Val1` and `msg_Val2` fields. The information corresponds directly to the `val1` and `val2` arguments the sending task specified when it called `SendSmallMsg()`. When you return such a message, you supply a 4-byte result code, and two new 32-bit values to put into `msg_Val1` and `msg_Val2`.

Working with Standard Messages

When you receive a standard message, the `msg_Val1` field of the message contains the address of a data area which you can access. The size of the data area is contained in `msg_Val2`. Typically, the data area points to a data structure belonging to the task that sent the message.

If the standard message comes from another task, as opposed to a sibling thread, the data area likely will only be readable by the receiving task, unless the sending task takes the extra steps to grant write permission to that data area. To determine whether your task can write to a data area pointed to in a standard message, use the `IsMemWritable()` function.

If you send messages between threads and tasks of the same application, it is often not necessary to go through the extra trouble of checking whether the data area is writable before writing to it, since this can be part of the protocol setup within your application.

When a task replies to a standard message, it provides a 4-byte result code, and new values for `msg_Val1` and `msg_Val2`. When a task calls `ReplyMsg()`, it can leave the values to what they were by simply using `msg_Val1` and `msg_Val2` as parameters to `ReplyMsg()`.

Working with Buffered Messages

When a task receives a buffered message, the `msg_Val1` field of the message contains the address of a read-only data area, and `msg_Val2` specifies the number of useful bytes within this buffer. The task can read this data at will.

When a task replies a buffered message, it supplies a 4-byte result code, and can optionally supply new data to be copied into the message's buffer. The task can determine the size available in the message's buffer by looking at the `msg_Val1Size` field of the message structure.

Pulling Back a Message

When a sending task sends a message, the task can pull the message out of a message queue before the message has been received. To do so, it uses the `GetThisMsg()` call:

```
Item GetThisMsg( Item msg )
```

The single argument, `msg`, specifies the message to rescind. If the message is still on the message port it was sent to, it is removed and its item number returned. If the message was already removed from the port by the receiving task, the function returns 0.

Replying to a Message

Once a message is received, the receiving task can reply, which returns a message to its reply port. The message contains four bytes that are set aside for a reply code; the replying task assigns an arbitrary value for storage there, often a value that reports the success or failure of message handling.

Once again, the three different message types require different handling when replying to a message.

Replying to Small Messages

To send back a small message in reply to the sending task, use this call:

```
Err ReplySmallMsg( Item msg, int32 result, uint32 val1,  
                  uint32 val2 )
```

`ReplySmallMsg()` accepts four arguments. The first argument, `msg`, is the item number of the message being returned in reply. The second argument, `result`, is a 32-bit value written into the `reply` field of the `Message` data structure. The third and fourth arguments, `val1` and `val2`, are data values written into the `msg_Val1` and `msg_Val2` fields of the `Message` data structure (just as they are in `SendSmallMsg()`). Note that no destination message port is specified because the message returns to its reply port, which is specified within the message data structure.

When `ReplySmallMsg()` executes, it sends a message back in reply and returns a value of 0 or greater if the call is successful, or an error code (a negative number) if the call is unsuccessful.

Replying to Standard and Buffered Messages

To send back a standard or buffered message in reply, use this call:

```
Err ReplyMsg( Item msg, int32 result, const void *dataptr, int32  
             datasize)
```

`ReplyMsg()` accepts four arguments. The first two, `msg` and `result`, are the same as those accepted by `ReplySmallMsg()`. The third, `dataptr`, is a pointer to a data block in memory. The fourth, `datasize`, is the size in bytes of that data block.

When `ReplyMsg()` executes, it sends the message in reply and returns 0 or greater if successful, or an error code (a negative number) if unsuccessful. The effect the reply has on the message depends on whether the message is standard or buffered, just as it does in `SendMsg()`.

If the message is standard, the data block pointer and the data size value are stored in the message, and are read as such by the task getting the reply. If the message is buffered, `ReplyMsg()` checks the size of the data block to see if it will fit in the message's buffer. If it doesn't fit, the call returns an error. If it does fit, `ReplyMsg()` copies the contents of the data block into the message's buffer and sends the message.

Messages With No Reply Ports

It is possible to create messages that don't have reply ports. When such a message is sent to a task, the ownership of the message is automatically transferred to the receiving task. This is a convenient feature allowing efficient uni-directional communications.

After the message has been sent, it becomes the responsibility of the receiving task or thread to delete that message once it is no longer useful. Alternatively, the message can be reused just as if it had been created by the receiving task to begin with.

Forwarding Messages to Another Port

When a task receives a message, it can forward it to another message port without doing any processing on it. This can be very useful. For example, imagine a server thread that has a public message port. Whenever commands come into the port, the server can spawn subthreads to handle the commands. After having started the subthread, the main server thread can forward the received message to the new subthread to have it perform the operation.

When a message ultimately gets replied by calling `ReplyMsg()` or `ReplySmallMsg()`, it will be sent back to the original creator of the message. Therefore, no matter how many tasks “touch” the message and simply forward it, when `ReplyMsg()` is called, the correct task will receive the reply.

Finding a Message Port

When a task sends a message to a message port, it needs to know the item number of the port. To get the item number of the port, it is often necessary to use the `FindMsgPort()` call:

```
Item FindMsgPort( const char *name )
```

When you provide `FindMsgPort()` the name of the message port to find, it returns the item number of that port. If the port doesn't exist, it returns a negative error code. If multiple ports of the same name exist, it returns the item number of the port with the highest priority.

Example Code for Messages

The code in Example 8-2 demonstrates how to send messages among threads or tasks.

The `main()` function of the program creates a message port where it can receive messages. It then spawns a thread. This thread creates its own message port and message. The thread then sends the message to the parent's message port. Once the parent receives the message, it sends it back to the thread.

Example 8-2 *Samples using the message passing routines (msgpassing.c).*

```
/* *****  
**  
**  @(#) msgpassing.c 95/10/15 1.6
```

```

**
*****/

#include <:kernel:types.h>
#include <:kernel:item.h>
#include <:kernel:task.h>
#include <:kernel:msgport.h>
#include <:kernel:operror.h>
#include <stdio.h>

/*****/

/* a signal mask used to sync the thread with the parent */
int32 parentSig;

/*****/

static void ThreadFunction(void)
{
    Item      childPortItem;
    Item      childMsgItem;
    Item      parentPortItem;
    Err       err;
    Message *msg;

    printf("Child thread is running\n");

    childPortItem = CreateMsgPort("ChildPort",0,0);
    if (childPortItem >= 0)
    {
        childMsgItem = CreateSmallMsg("ChildMsg",0,childPortItem);
        if (childMsgItem >= 0)
        {
            parentPortItem = FindMsgPort("ParentPort");
            if (parentPortItem >= 0)
            {
                /* tell the parent we're done initializing */
                SendSignal(CURRENTTASK->t_ThreadTask->t.n_Item,parentSig);

                err = SendSmallMsg(parentPortItem,childMsgItem,12,34);
                if (err >= 0)
                {

```

```
err = WaitPort(childPortItem, childMsgItem);
if (err >= 0)
{
    msg = MESSAGE(childMsgItem);
    printf("Child received reply from parent: ");
    printf("msg_Result %d, msg_DataPtr %d, msg_DataSize %d\n",
        msg->msg_Result, msg->msg_DataPtr,
        msg->msg_DataSize);
}
else
{
    printf("WaitPort() failed: ");
    PrintfSysErr(err);
}
}
else
{
    printf("SendSmallMsg() failed: ");
    PrintfSysErr(err);
}

SendSignal(CURRENTTASK->t_ThreadTask->t.n_Item, parentSig);
}
else
{
    printf("Could not find parent message port: ");
    PrintfSysErr(parentPortItem);
}
DeleteMsg(childMsgItem);
}
else
{
    printf("CreateSmallMsg() failed: ");
    PrintfSysErr(childMsgItem);
}
DeleteMsgPort(childPortItem);
}
else
{
    printf("CreateMsgPort() failed: ");
    PrintfSysErr(childPortItem);
}
}
```

/*****

```
int main(void)
{
    Item      portItem;
    Item      threadItem;
    Item      msgItem;
    Message *msg;

    parentSig = AllocSignal(0);
    if (parentSig > 0)
    {
        portItem = CreateMsgPort("ParentPort",0,0);
        if (portItem >= 0)
        {
            threadItem = CreateThread(ThreadFunction, "Child", 0, 2048, NULL);
            if (threadItem >= 0)
            {
                /* wait for the child to be ready */
                WaitSignal(parentSig);

                /* confirm that the child initialized correctly */
                if (FindMsgPort("ChildPort") >= 0)
                {
                    printf("Parent waiting for message from child\n");

                    msgItem = WaitPort(portItem,0);
                    if (msgItem >= 0)
                    {
                        msg = MESSAGE(msgItem);
                        printf("Parent got child's message: ");
                        printf("msg_DataPtr %d, msg_DataSize %d\n",
                            msg->msg_DataPtr, msg->msg_DataSize);
                        ReplySmallMsg(msgItem,56,78,90);
                    }
                    else
                    {
                        printf("WaitPort() failed: ");
                        PrintfSysErr(msgItem);
                    }
                }
            }

            /* wait for the thread to tell us it's done before we zap it */
            WaitSignal(parentSig);

            DeleteThread(threadItem);
        }
        else
    }
```



```
        {
            printf("CreateThread() failed: ");
            PrintfSysErr(threadItem);
        }
        DeleteMsgPort(portItem);
    }
    else
    {
        printf("CreateMsgPort() failed: ");
        PrintfSysErr(portItem);
    }
    FreeSignal(parentSig);
}
else
{
    printf("AllocSignal() failed\n");
}

return 0;
}
```

The Portfolio I/O Model

This chapter provides an overview of the 3DO system's hardware devices and the system software that controls them.

This chapter contains the following topics:

Topic	Page Number
Introduction	101
Hardware Devices and Connections	102
I/O Architecture	103
Performing I/O	105

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Introduction

The 3DO environment is rich in device possibilities: any 3DO unit may be set up with few or many devices. Manufacturers can make 3DO units with a bare-bones configuration, or they can add extra devices such as a second CD-ROM drive or a digital television tuner. Once the 3DO unit is out of the box, users can add more devices by plugging in any of a large variety of controls, from simple control pads to keyboards, or by adding peripherals such as modems or RAM mass storage.

Because a task can run on a machine with any number of devices attached to it, a task can't assume that it knows what devices are available to it. Assume, for example, that a task relied on the low-level hardware details of a particular CD-ROM drive. If a manufacturer of 3DO suddenly changed the drive's hardware, the task might suddenly fail to operate correctly.

To help a task traverse a world of devices that may change from minute to minute, Portfolio provides device drivers and I/O calls that can sense attached devices and know how to communicate with them, all without specific hardware knowledge on the part of client tasks.

Hardware Devices and Connections

The 3DO standard defines a set of buses and ports for connecting hardware devices. These bus and port definitions make the best use of existing hardware technologies.

Warning: *Keep in mind a very important warning as you read about the hardware described in this chapter: the bus and port definitions may change.*

As hardware prices fall, better performance will be possible using advanced hardware that costs less than the currently specified hardware; the 3DO standard *will* change to take advantage of better hardware as it becomes available. In other words, the hardware descriptions in this chapter are a snapshot of the current 3DO standards. Don't write software that assumes 3DO hardware is always going to be manufactured as it is now. Fortunately, you don't have to write hardware-specific code, because Portfolio takes care of the specifics for you, as you'll read later in this chapter.

The Control Port

The control port, which is a serial port, is used mainly to connect user-interface devices that control the 3DO unit. These devices include:

- ◆ The **control pad** is a standard control that includes a joystick and buttons, and may include a headphone jack. This device is used to point, select, and control on-screen action. It can also provide stereo sound accompaniment through a headphone jack.
- ◆ The **photo-optic gun** is used to shoot at targets on the video screen.
- ◆ **Stereoscopic glasses** are used to view stereoscopic displays.
- ◆ A **mouse or trackball** is used to point to and select objects on the screen.
- ◆ A **keyboard** is used to type in text.
- ◆ Any number of **custom devices** created and sold by 3DO-licensed manufacturers.

The control port supports a daisy chain of devices. Because a new device can always be plugged in to the last device in the chain, the control port doesn't have a fixed maximum number of devices, but instead is limited by the data bandwidth of the port. In other words, you can keep plugging in more control devices until the control port is overwhelmed with data and starts to choke up. If this happens, the control port fails to recognize the newly plugged in control devices that go beyond the port's capacities.

Tasks interface to the control port using the event broker. To learn about interfacing to external control port hardware, see Chapter 13, "The Event Broker".

I/O Architecture

Portfolio provides services to access hardware devices in a hardware-independent manner. This allows applications to remain compatible across a wide range of hardware products. The Portfolio I/O model is a central component that enables a wide range of exotic devices to coexist and evolve for many years.

The Portfolio I/O model is inherently asynchronous. Lengthy requests made by a task to a hardware device are handled in the background. The client task receives a notification when the request has been satisfied. This architecture allows a single task to have multiple outstanding requests pending on different hardware devices, which tends to maximize the use of the system bandwidth by keeping all subsystems working.

Imagine a task that needs to read a block of data from a CD-ROM. It sends a request to the CD-ROM device driver. The driver does the setup necessary to initiate a hardware transfer. The CD-ROM mechanism is then activated and the needed block of data is transferred to memory. When the transfer is complete, an interrupt occurs that wakes up the driver. The driver then notifies the client task that the data it needs has arrived. While the hardware is servicing the request from the task, the task is free to do other activities, such as playing music.

The Portfolio I/O model relies on three basic abstractions: the *device*, the *driver*, and the *I/O request*.

Devices

A software device may or may not correspond to a hardware device. Technically, a software device is anything that responds to I/O requests. A software device might correspond directly to a hardware device such as a timer, a CD-ROM player, or a control pad; it might correspond to a port or bus that controls many different devices; or it might correspond to a more abstract entity such as an open file on a CD-ROM disk. Portfolio treats a software device as a single I/O mechanism, no matter what the device's correspondence to actual hardware.

Note: Whenever you read the term "device" by itself, it refers to a software device. A hardware device is always referred to as "hardware device."

Portfolio maintains a list of all devices resident or active on the 3DO unit. This list changes as devices are added to or taken from the system. Each device in the list has an item number; a task that requests I/O with a device must refer to the device by its item number. To find a device's item number, a task uses the call `OpenDeviceStack()`, where the task identifies the device it wants and the kernel returns the device's item number.

Device Stacks

Some devices can operate in isolation; that is, a single device can perform useful services. But some devices depend on other devices. For example, an external modem may be driven by a *modem device*. This modem device may deal only with the modem command set, and require a *serial port* device to do the job of actually sending data out the serial port to the modem. In this case, the modem device is said to be *stacked* on top of the serial port device. Generally, client software does not need to know or care about the details of the device stack. Client software interfaces directly to the top device in the stack; everything that happens below that is invisible to the client.

Drivers

Every device maintained by the kernel must have a corresponding driver before any task can use the device. The driver accepts and acts on every task request for I/O. A single driver may have more than one device active at one time.

I/O Requests (IOReqs)

An I/O request (IOReq for short) is a data structure that the kernel passes back and forth between a task and a device. Think of the IOReq as a messenger that goes from task to device, passing along an I/O command and other I/O information. When an I/O operation is finished, the device returns the IOReq to the task, passing along any reports it has to make on the I/O operation.

Before a task can communicate with a device, the task must create an IOReq using the `CreateIOReq()` call. An IOReq is a system item that must be used to communicate with a device.

Within an IOReq lies information necessary for executing an I/O command. Part of that information is the `IOInfo` data structure, which contains an I/O command, a unit specification, pointers to read and write buffers, and other I/O parameters provided by the task requesting I/O.

To initiate an I/O operation, a task must give the `IOReq` to a device by calling `SendIO()` or `DoIO()`. When the device completes the operation, it notifies the calling task that the operation is complete. The calling task is then free to use the `IOReq` to initiate further I/O operations.

A task gets notified that an I/O operation is complete by either receiving a signal or receiving a message. The notification mechanism is determined when the `IOReq` is created.

Performing I/O

From a task's point of view, the process of I/O between a task and a device can be broken down into four stages:

- ◆ **Preparing for I/O.** The task opens a device, creates one or more `IOReq` data structures, and creates one or more `IOInfo` data structures.
- ◆ **Sending an `IOReq`.** The task fills in `IOInfo` values, writes output data into a write buffer (if appropriate), and submits the `IOInfo` values to the kernel for processing.
- ◆ **Receiving an `IOReq`.** The task receives notification that an I/O operation is complete, reads data about the operation within a returned `IOReq`, and (if appropriate) reads data from a read buffer.
- ◆ **Finishing I/O.** The task finishes exchanging `IOReq` structures with a device, deletes the `IOReq` structures it created earlier, and closes the device.

If a task wants to perform a series of I/O operations on a single device, it only needs to prepare for I/O once. The task can then send and receive `IOReq` structures to and from the device until it is finished.

Preparing for I/O

Before a task can send and receive `IOReq` structures, it must prepare for I/O by opening a device and creating `IOReq` and `IOInfo` data structures.

The first job the client must perform is to identify the device stack to be opened. The client does this by calling `CreateDeviceStackList()`. The client passes arguments to this function which identify the functional requirements of the device stack. For example, it may be required that the device support a certain set of device commands. The `CreateDeviceStackList()` function returns a list of all device stacks that satisfy these requirements. If there is more than one device stack in the list, the client must choose which one to use. Depending on the situation, the client can simply choose a device stack at random (if all are equally acceptable), or may need to ask the user to help in the selection.

In any case, once the device stack is selected, the client then calls `OpenDeviceStack()`. When `OpenDeviceStack()` executes, it opens the device stack and re-

turns the item number of the top-most device. Future I/O calls to the device use the device's item number.

Creating an IOReq

Once a task opens a device, the task must create one or more IOReq items to communicate with the device.

Deciding How to Notify the Task - Signal or Message

Before creating an IOReq, it's important to consider how the task will be notified when the IOReq is returned. There are two options:

- ◆ Notification by signal
- ◆ Notification by message

Notification by signal is the default method. When the device finishes with an I/O operation and wants to return the IOReq to the task, it sends the system signal `SIGF_IODONE` to the task. The task then knows that there is a completed IOReq, and it can retrieve the IOReq at any time.

Notification by message is specified by providing a message port when an IOReq is created. When the device finishes with an I/O operation and wants to return the IOReq, it sends a message to the specified port. The task can read the message to get a pointer to the IOReq and then read or reuse the IOReq when it wants to.

Each notification method has advantages and disadvantages.

- ◆ Notification by signal uses fewer resources, but doesn't identify the returning IOReq. It merely says that an IOReq has returned. Notification by signal is useful for I/O operations that use a single IOReq passed back and forth between the task and a device.
- ◆ Notification by message uses slightly more resources, but because each message identifies a particular IOReq, the task knows exactly which IOReq is returned when it receives notification. Notification by message is useful for I/O operations that use more than one IOReq.

There is a way to get the advantages of both methods by using signals to identify the returning IOReq. To do this you specify the `CREATEIOREQ_TAG_SIGNAL` tag with a signal mask indicating the desired signal(s) when creating the IOReq. The system will then return the signal(s) that you specified instead of the default `SIGF_IODONE` signal. The procedure for doing this is described in the next subsection.

Creating the IOReq

Once a task opens a device, the task creates an IOReq data structure using this call:

```
Item CreateIOReq( const char *name, uint8 pri, Item dev, Item mp )
```

The *name* argument is a pointer to a null-terminated character string containing the name of this IOReq. If the IOReq is unnamed, this argument should be NULL. The *pri* argument is a priority value from a low priority of 0 to a high priority of 255. The *dev* argument is the item number of the opened device to which this IOReq is to be sent.

The *mp* argument is the item number of a message port where the task is notified of the IOReq's completion. If the task wants to be notified of IOReq completion by message, it must create a message port and supply its item number here. If the task wants only to be notified of IOReq completion by signal, then this argument should be set to zero.

When the `CreateIOReq()` call executes, the kernel creates an IOReq to be sent to the specified device and returns the item number of the IOReq. All future I/O calls using this IOReq specify it with its item number.

If you want to use signals to identify the returning IOReq, create and use a function like the `CreateCustomSignalIOReq()` function shown below instead of the standard `CreateIOReq()` function. `CreateCustomSignalIOReq()`, which is not in the API, takes a signal mask parameter instead of a message port. When you use this call to create an IOReq structure, the signals you supply are sent to your task when the I/O operation completes.

```
Item CreateCustomSignalIOReq(const char *name, uint8 pri, Item dev,
                             int32 sigMask)
{
    return CreateItemVA(MKNODEID(KERNELNODE, IOREQNODE),
        TAG_ITEM_PRI,    pri,
        (name ? TAG_ITEM_NAME : TAG_NOP), name,
        CREATEIOREQ_TAG_DEVICE,    dev,
        CREATEIOREQ_TAG_SIGNAL,    sigMask,
        TAG_END);
}
```

Initializing an IOInfo Structure

Although a task can create an IOReq, it can't directly set values in it because an IOReq is an item. The task instead defines an IOInfo data structure in its own memory, and then, when it requests an I/O operation, the task fills in the appropriate values and passes the IOInfo on to the kernel with the IOReq.

The definitions of the IOInfo data structure and the IOBuf data structure used within it (both defined in `<kernel/io.h>`) are as follows:

Example 9-1 *IOInfo, IOBuf structures*

```
/* Information supplied by the user when initiating an I/O operation */

typedef struct IOInfo
{
    uint32 ioi_Command;    /* command to be executed */
    uint32 ioi_Flags;      /* control flags */
    uint32 ioi_CmdOptions; /* device dependent options */
    int32 ioi_Offset; /* offset into device for transfer to begin */
    IOBuf ioi_Send;        /* information being written out */
    IOBuf ioi_Recv;        /* where to put incoming information */
    void *ioi_UserData;    /* user-private data */
} IOInfo;

typedef struct IOBuf
{
    void *iob_Buffer; /* ptr to user's buffer */
    int32 iob_Len; /* len of this buffer, or transfer size */
} IOBuf;
```

The possible values for the `IOInfo` fields depend on the device to which the `IOReq` is sent. The fields include the following:

- ◆ **ioi_Command** carries the I/O command to be executed by the device driver.
- ◆ **ioi_Flags** contains flags for I/O operation variations. Portfolio currently defines just one flag: `IO_QUICK`, which asks that I/O take place immediately (synchronously) without notification. All other bits must be set to 0.
- ◆ **ioi_CmdOptions** sets command options that vary from device to device.
- ◆ **ioi_UserData** contains a pointer to anything the task wishes to point to. This is a handy field for keeping a reminder of the data on which an I/O operation is working. When the I/O operation completes, you can then look in the `io_Info.ioi_UserData` field of the `IOReq` structure to retrieve this pointer.
- ◆ **ioi_Offset** contains an offset into the device—a location—where the I/O operation begins. For output operations, this offset defines the starting point in the device where the task's data is written. For input operations, this offset defines the starting point in the device where data is read. This value, when supported by a device, uses the units of block size for whatever device it is addressing.

- ◆ **ioi_Send** contains the pointer to and size of a write buffer in which the task stores output data to be written to the device.
- ◆ **ioi_Recv** contains the pointer to and size of a read buffer in which the driver stores input data read from the device.

The **IOInfo** structure should always contain a command value in **ioi_Command**. If the command involves writing data, **ioi_Send** should be filled in with a write buffer definition; if the command involves reading data, **ioi_Recv** should be filled in with a read buffer definition. And if the task wants the fastest possible I/O operation with a device that can respond immediately (a timer, for example), **ioi_Flags** should be set to **IO_QUICK**. Any other fields should be filled in as appropriate for the operation.

Passing IOInfo Values to the Device

To pass the **IOInfo** values to the system and send the **IOReq**, the task can use the **SendIO()** function:

```
Err SendIO( Item ioReqItem, const IOInfo *ioiP )
```

The *ioReqItem* argument is the item number of the **IOReq** to be sent to the device. The *ioiP* argument is a pointer to the **IOInfo** structure that describes the I/O operation to perform.

When the kernel carries out **SendIO()**, it copies the **IOInfo** values into the **IOReq**, then checks the **IOReq** to be sure that all values are appropriate. If they are, the kernel passes on the I/O request to the device responsible. The device then carries out the request.

SendIO() returns 1 if immediate I/O was used and the **IOReq** is immediately available to the task. **SendIO()** returns a 0 if I/O was done asynchronously, which means that the request is being serviced by the device in the background. **SendIO()** returns a negative error code if there was an error in sending the **IOReq**. This usually occurs if there were inappropriate values included in the **IOInfo** structure.

Asynchronous and Synchronous I/O

When a task sends an I/O request to a device using **SendIO()**, the device may or may not satisfy the request immediately. If **SendIO()** returns 1, it means that the operation is completed and no other actions are expected. If **SendIO()** returns a 0, it means that the I/O operation has been deferred and is being worked on in the background.

When an operation is deferred, your task is free to continue executing while the I/O is being satisfied. For example, if the CD-ROM device is doing a long seek operation in order to get to a block of data you have asked it to read, you can continue executing the main loop of your task while you wait for the block to be transferred.

When `SendIO()` returns 0, which means the I/O request is being serviced asynchronously, you must wait for a notification that the I/O operation has completed before you can assume anything about the state of the operation. As shown previously, when you create an `IOReq` using `CreateIOReq()`, you can specify one of two types of notification: signal or message. When an asynchronous I/O operation completes, the device sends your task a signal or a message to inform you of the completion.

Once you receive the notification that an I/O operation is complete, you must call `WaitIO()` to complete the I/O process.

```
Err WaitIO( Item ioreq )
```

`WaitIO()` cleans up any loose ends associated with the I/O process. `WaitIO()` can also be used when you wish to wait for an I/O operation to complete before proceeding any further. The function puts your task or thread in wait state until the specified `IOReq` has been serviced by the device.

The return value of `WaitIO()` corresponds to the result code of the whole I/O operation. If the operation fails for some reason, `WaitIO()` returns a negative error code describing the error.

If you have multiple I/O requests that are outstanding, and you receive a signal telling you an I/O operation is complete, you might need to determine which I/O request is complete. Use the `CheckIO()` function:

```
int32 CheckIO( Item ioreq )
```

`CheckIO()` returns 0 if the I/O operation is still in progress. It returns greater than 0 if the operation completes; it returns a negative error code if something is wrong.

Do not use `CheckIO()` to poll the state of an I/O request. Use `WaitIO()` if you need to wait for a specific I/O operation to complete, or use `WaitSignal()` or `WaitPort()` if you must wait for one of a number of I/O operations to complete.

There are many cases where an I/O operation is very short and fast. In these cases, the overhead of notifying your task when the I/O completes becomes significant. The I/O subsystem provides a quick I/O mechanism to help remove this overhead as much as possible.

Quick I/O occurs whenever `SendIO()` returns 1. It tells you that the I/O operation is complete, and that no signal or message will be sent to your task. You can request quick I/O by setting the `IO_QUICK` bit in the `ioi_Flags` field of the

`IOInfo` structure before you call `SendIO()`. `IO_QUICK` is merely a request for quick I/O. It is possible that the system cannot perform the operation immediately. Therefore, check the return value of `SendIO()` to make sure the I/O was done immediately. If it was not done synchronously, you have to use `WaitIO()` to wait for the I/O operation to complete.

The fastest and simplest way to do quick I/O is to use the `DoIO()` function:

```
Err DoIO( Item ioreq, const IOInfo *ioInfo )
```

`DoIO()` works just like `SendIO()`, except that it guarantees that the I/O operation is complete once it returns. You do not need to call `WaitIO()` or `CheckIO()` if you use `DoIO()` on an `IOReq`. `DoIO()` always requests quick I/O, and if the system is unable to do quick I/O, this function automatically waits for the I/O to complete before returning.

Completion Notification

When an I/O operation is performed asynchronously, the device handling the request always sends a notification to the client task when the I/O operation is complete. As mentioned previously, the notification can be either a signal or a message.

When you create `IOReq` items with signal notification, the responsible devices send your task the `SIGF_IODONE` signal whenever an I/O operation completes. If you have multiple I/O requests outstanding at the same time, you can use the following to wait for any of the operations to complete.

```
WaitSignal(SIGF_IODONE);
```

When `WaitSignal()` returns, you must call `CheckIO()` on all of the outstanding I/O requests you have to determine which one is complete. Once you find a completed request, call `WaitIO()` with that `IOReq` to mark the end of the I/O operation.

If you created your `IOReq` structures with message notification, you will receive a message whenever an I/O operation completes. The message is posted to the message port you specified when you created the `IOReq`. If you have multiple message-based I/O requests outstanding, you could wait for them using:

```
msgItem = WaitPort(replyPort, 0);
```

`WaitPort()` puts your task to sleep until any of the `IOReq` structures complete. Once `WaitPort()` returns, you can look at three fields of the message structure to obtain information about the `IOReq` that has completed:

Example 9-2 *Looking at fields of the message structure*

```
{
Item msgItem;
Msg *msg;

    msg      = (Msg *) LookupItem(msgItem);
    ioreq    = (Item) msg->msg_DataPtr;
    result   = (Err) msg->msg_Result;
    user     = msg->msg_DataSize;
}
```

The `msg_DataPtr` field contains the item number of the `IOReq` that has completed. The `msg_Result` field contains the result code of the I/O operation. This is the same value that `WaitIO()` would return for this `IOReq`. Finally, `msg_DataSize` contains the value of the `ioi_UserData` field from the `IOInfo` structure used to initiate the I/O operation with `SendIO()`.

Reading an IOReq

Once an `IOReq` returns to a task, the task can read the `IOReq` for information about the I/O operation. To read an `IOReq`, a task must get a pointer to the `IOReq` using the `IOREQ()` call (actually a macro that invokes `LookupItem()`). For example:

```
ior = IOREQ(ioreqitem)
```

Once a task has the address of an `IOReq`, it can read the values in the different fields of the `IOReq`. An `IOReq` data structure is defined as follows:

Example 9-3 *IOReq structure*

```
typedef struct IOReq
{
    ItemNode      io;
    struct Device *io_Dev;          /* Device this IOReq belongs to */
    IOInfo        io_Info;         /* as supplied by user */
    int32         io_Actual;        /* actual size of request completed */
    uint32        io_Flags;        /* state flags, see below */
    int32         io_Error;        /* completion status of last use */
    int32         io_Extension[2]; /* device-specific data */

    union
    {
        Item      io_msgItem;      /* message sent on completion */
    };
};
```

```
int32      io_signal;      /* or signal sent on completion */
} io_Completion;

uint32      io_Private0[5];
} IOReq;

#define io_MsgItem io_Completion.io_msgItem
#define io_Signal io_Completion.io_signal
```

Fields that offer information to the average task are:

- ◆ `io_Info.ioi_UserData`. A copy of the `ioi_UserData` field from the `IOInfo` structure supplied by the task when calling `SendIO()` or `DoIO()`. This is a convenient location for the task to store context information associated with the `IOReq`.
- ◆ `io_Actual`. Supplies the size in bytes of the I/O operation just completed. This is a convenient place to find out the size of the last I/O transfer.
- ◆ `io_Error`. Contains any errors generated by the device carrying out the I/O operation. The meaning of this value depends on the device that was used. This value is also returned by `DoIO()` and `WaitIO()`.

Continuing I/O

Tasks often have a series of I/O operations to carry out with a device. If so, a task can recycle an `IOReq` once it's been returned; the task supplies new values in an `IOInfo` data structure, and then uses `SendIO()` or `DoIO()` to request a new operation.

A task is not restricted to a single `IOReq` for a single device. A task can create two or more `IOReq` structures for a device, and work with one `IOReq` while others are sent to the device or are awaiting action by the task. For example, use `DoIO()` on image files, then `SendIO()` on sound files while processing the image files.

Aborting I/O

If an asynchronous I/O operation must be aborted while in process at the device, the issuing task can use the `AbortIO()` function:

```
Err AbortIO( Item ioreq )
```

`AbortIO()` accepts the item number of the `IOReq` that is responsible for the operation to be aborted. When executed, it notifies the device that the operation should be aborted. When it is safe, the operation is aborted and the device sets the `IOReq`'s `io_Error` field to `ABORTED`. The device then returns the `IOReq` to the task.

A task should always follow the `AbortIO()` call with a `WaitIO()` call so the task will wait for the abort to complete and the `IOReq` to return.

Finishing I/O

When a task completely finishes I/O with a device, the task should clean up by deleting any `IOReq` structures it created and by closing the device. To delete an `IOReq`, a task uses the `DeleteIOReq()` function:

```
Err DeleteIOReq( Item ioreq )
```

This call accepts the item number of an `IOReq` and frees any resources the `IOReq` used.

To close a device, a task uses the `CloseDeviceStack()` call:

```
Err CloseDeviceStack( Item device )
```

Portfolio Devices

This chapter lists the standard Portfolio devices and their associated commands and options.

This chapter contains the following topics:

Topic	Page Number
Introduction	115
Timer Devices	117
File Devices	123
Serial Devices	131

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Introduction

Some of the devices currently in the 3DO system include:

- ◆ Timer device
- ◆ File devices
- ◆ Serial devices
- ◆ CD-ROM device (not documented here)
- ◆ Control port device (not documented here)
- ◆ Microslot device (not documented here)
- ◆ MPEG video device (not documented here)

All devices use I/O request items for communicating information between tasks and the device. Refer to Chapter 9, "The Portfolio I/O Model" for more information on devices and I/O requests.

You can access 3DO devices in two ways:

- ◆ You can use a function call provided for the desired purpose. In this case, you call the appropriate function, passing the necessary parameters. The function is provided in one of the Portfolio folios.
- ◆ You can create an IOReq yourself and pass it to the device.

Normally, it is easier to use a function call if one is provided, but some operations can only be done by sending an IOReq manually.

To pass an IOReq, the task must first obtain a handle on a Device Item, which comes from either calling `OpenDeviceStack()` or `OpenFile()`.

A task must also get its own IOReq and IOInfo data structures to pass information to and from the device. The IOReq data structure is usually created using the call `CreateIOReq()` although some devices, such as timers, provide their own calls to create an IOReq. The IOInfo data structure is allocated by the task either via a memory allocation call or on the stack.

Note: *Make sure that all bits in the IOInfo structure are zeroed out before you use the structure.*

After setting the IOInfo fields, a task uses the `SendIO()` or `DoIO()` functions to pass the item number of the IOReq structure and a pointer to the IOInfo structure to the device. The IOReq is passed to the device, and the appropriate action is taken by the device as specified by the IOInfo structure fields.

A task sends a command to a device in the IOInfo data structure. The `IOInfo.ioi_Command` field is set to the command (e.g., `FILECMD_FSSTAT`) that the task intends to send to the device. In addition to the command, some devices allow the task to specify options to the command. The options are set up by defining the `IOInfo.ioi_CmdOptions` field. The `IOInfo.ioi_Send` and `IOInfo.ioi_Recv` substructures must also be set for the information to be sent to the device and to be received back from the device. The sections that follow provide more detailed information about the commands and command options, and the send and receive buffers that a task uses to communicate with a device.

You can use either method to accomplish many device operations. Other operations do not have a function call and require you to use the manual IOReq procedure. In the following sections, the function call will be described if one exists. The manual IOReq method will only be described if it is the easiest or only way to accomplish the operation.

Timer Devices

Timer devices are software components that provide a standardized interface to the timing hardware.

There are two separate clocks that offer different timing characteristics: the microsecond clock and the vertical blank clock. The microsecond clock deals with time quantities using seconds and microseconds, while the vertical blank clock counts time in vertical blank intervals. Vertical blank intervals are discussed in more detail later in this chapter. Both clocks respond to the same types of commands.

Using either of the clocks, you can do four basic operations:

- ◆ Sample the current time
- ◆ Ask the timer to notify you when a given time arrives
- ◆ Ask the timer to notify you after a given amount of time passes
- ◆ Ask the timer to notify you at regular intervals until you ask it to stop

The following sections describe how to perform these operations with both timer device clocks.

Creating a Timer IOReq

Use the `CreateTimerIOReq()` function to create an `IOReq` for use with either the microsecond clock or the vertical blank clock. `CreateTimerIOReq()` creates and opens the device stack and creates the `IOReq`.

```
Item CreateTimerIOReq(void)
```

This function returns either an `IOReq` item or a negative error code.

The Microsecond Clock

The microsecond clock provides very high-resolution timing. Although its short-term accuracy is high, its time base can drift slightly over extended periods of time.

The microsecond clock deals in time quantities using the `TimeVal` structure:

Example 10-1 *TimeVal structure*

```
typedef struct timeval
{
    int32 tv_Seconds;           /* seconds */
    int32 tv_Microseconds;     /* and microseconds */
} TimeVal;
```

Reading the Current System Time

To read the current system time, call `SampleSystemTimeTV()`.

```
void SampleSystemTimeTV ( TimeVal *time )
```

This function returns, in the `TimeVal` structure, the time in seconds and microseconds since the machine was last booted. This is the same result that you would get by sending an `IOReq` manually with the `ioInfo.ioi_Command` field set to `TIMERCMD_GETTIME_USEC`.

Waiting for a Specific Time to Arrive

A common use for the timer device is to provide automatic notification of the passage of time. For example, if you want a given picture to remain on the display for exactly 1 second, you can send a command to the timer device telling it to send you a signal when 1 second has passed. While you are waiting for that second to pass, your task can do other work, such as play music, confident in the fact that the timer device will notify it when the appropriate amount of time has passed.

You can ask the timer device to notify you when a specific time arrives. To do this, you must first ask the system what the current time is by calling `SampleSystemTimeTV()`. Once you know the current time, you can use the `AddTimes()` and `SubTimes()` calls to calculate the time to receive a notification.

Once you have calculated the time to be notified, you can use the `WaitUntil()` function, or you can send an `IOReq` with the `TIMERCMD_DELAYUNTIL_USEC` command to the timer device. The two methods are not equivalent because the `WaitUntil()` call is synchronous (the task must wait until it completes), but sending an `IOReq` can be done asynchronously (the task can do other work while waiting).

Using `WaitUntil()`

You first create an `IOReq`, using `CreateTimerIOReq()` and then call `WaitUntil()`, passing the `IOReq` and the time to wait until (in seconds and microseconds).

```
Err WaitUntil(Item ioreq, uint32 seconds, uint32 micros)
```

The calling task is put to sleep until the specified time is reached.

Sending an `IOReq` with `TIMERCMD_DELAYUNTIL_USEC`

First, use `CreateTimerIOReq()` to create an `IOReq`.

Then you must initialize the `IOInfo` structure in the following way:

Example 10-2 *Initializing IOInfo for TIMERCMD_DELAYUNTIL_USEC*

```
IOInfo  ioInfo;
TimeVal tv;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = TIMERCMD_DELAYUNTIL_USEC;
ioInfo.ioi_Send.iob_Buffer = &tv;
ioInfo.ioi_Send.iob_Len  = sizeof(tv);
```

The `ioi_Command` field is set to `TIMERCMD_DELAYUNTIL_USEC`, indicating that the timer will wait until a specific time arrives. `ioi_Send.iob_Buffer` points to a `TimeVal` structure which contains the amount of time to wait. Finally, `ioi_Send.iob_Len` holds the size of the `TimeVal` structure.

You can send the I/O request to the timer device using either `DoIO()` or `SendIO()`. When using `DoIO()`, your task is put to sleep until the requested time. If you use `SendIO()`, then your task is free to continue working while the timer is counting time. When the requested time arrives, the timer device either sends the task the `SIGF_IODONE` signal, or sends a message as specified in the I/O request.

Waiting a Specific Amount of Time

Instead of asking the timer to wait until a given time, you can tell it to wait for a fixed interval of time to pass. To achieve this, follow the procedure in the previous section except that you use `WaitTime()` or send an `IOReq` with the `IOInfo` structure initialized differently.

To use `WaitTime()`, create an `IOReq` and then call `WaitTime()`, passing the `IOReq` and the length of the desired interval in seconds and microseconds.

```
Err WaitTime(Item ioreq, uint32 seconds, uint32 micros)
```

The calling task is put to sleep for the specified interval of time.

To send an `IOReq` manually, set the `ioInfo.ioi_Command` field to `TIMERCMD_DELAY_USEC`, and the `TimeVal` structure to the desired time interval in seconds and microseconds. Then invoke `DoIO()` (synchronous) or `SendIO()` (asynchronous) as described previously.

Getting Repeated Notifications - Metronome Timing

Sometimes you will want to have the timer automatically generate a signal at repeated fixed intervals. The easiest way to do this is to create an `IOReq` and then call `StartMetronome()`, passing the `IOReq` item, the desired time interval in seconds and microseconds, and a signal mask indicating the signal that you want the timer to send to the calling task at the end of each time interval.

```
Err StartMetronome(Item ioreq, uint32 seconds, uint32 micros,
                    int32 signal)
```

To stop, the timer from sending these signals, call `StopMetronome()`, passing the `IOReq` item.

```
Err StopMetronome(Item ioreq)
```

Note that you could also start a metronome interval by creating and sending an `IOReq` with the `ioInfo.ioi_Command` field set to `TIMERCMD_METRONOME_USEC`.

The Vertical Blank Clock

The vertical blank clock provides a fairly coarse measure of time, but is very stable over long periods of time. It offers a resolution of either 1/60th of a second on NTSC systems or 1/50th of a second on PAL systems. Vertical blanking is a characteristic of raster scan displays, and occurs on a fixed time-scale synchronized with the display hardware.

A *vblank* is the amount of time it takes for the video beam to perform an entire sweep of the display. Given that displays operate at different refresh rates in NTSC (60 Hz) compared to PAL (50 Hz), the amount of time taken by a vblank varies. Since the vblank clock of the timer device deals with time exclusively in terms of vblank units, waiting for a fixed number of vblanks takes different amounts of time on NTSC and PAL.

The advantages of the vertical blank clock are that it remains stable for very long periods of time; it involves slightly less overhead than the microsecond unit; and it is synchronized with the video beam. Being synchronized with the video beam is very important when creating animation sequences.

The vertical blank clock of the timer device deals in time quantities using the `TimeValVBL` type, which is defined as:

```
typedef uint64 TimeValVBL;
```

It is important to understand that the timer counts vblanks in a very strict way. Whenever the video beam reaches a known location on the display, the vblank counter is incremented. So if you ask the timer to wait for 1 vblank when the beam is near the trigger location, the I/O request will be returned in less than 1/60th or 1/50th of a second.

Reading the Current Vblank Count

To read the current vblank count, call `SampleSystemTimeVBL()`.

```
void SampleSystemTimeVBL ( TimeValVBL *tv )
```

This function returns, in the `TimeValVBL` variable, the current vblank count, which is the number of vblanks since the machine was last booted. This is the same result that you would get by sending an `IOReq` manually with the `ioInfo.ioi_Command` field set to `TIMER_CMD_GETTIME_VBL`.

Waiting for a Specific Vblank Count

Similar to the microsecond clock, the vblank clock can wait for a specific time -- expressed in terms of vblanks.

You can use the `WaitUntilVBL()` function, which works synchronously, or you can send an `IOReq` manually with `ioInfo.ioi_Command` set to `TIMERCMD_DELAYUNTIL_VBL`, which can work synchronously (if initiated by `DoIO`) or asynchronously (if initiated by `SendIO`).

Call `WaitUntilVBL()` as follows, passing the `IOReq` (obtained by executing `CreateTimerIOReq()`) and the vblank count that must be reached.

```
Err WaitUntilVBL(Item ioreq, uint32 fields)
```

The task sleeps until the specified vblank count is reached.

When sending an `IOReq` manually, initialize the `IOInfo` structure as follows.

Example 10-3 *Initializing IOInfo for TIMERCMD_DELAYUNTIL_VBL*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = TIMERCMD_DELAYUNTIL_VBL;
ioInfo.ioi_Offset  = vblankCountToWaitUntil;
```

Then use `DoIO()` or `SendIO()` to send the request.

Waiting for an Interval (Number of Vblanks)

The vblank unit can also wait for a given number of vblanks, that is, a specific interval of time -- expressed in vblanks.

You can use the `WaitTimeVBL()` function, which works synchronously, or you can send an `IOReq` manually with `ioInfo.ioi_Command` set to `TIMERCMD_DELAY_VBL`, which can work synchronously (if initiated by `DoIO`) or asynchronously (if initiated by `SendIO`).

Call `WaitTimeVBL()` as follows, passing the `IOReq` (obtained by executing `CreateTimerIOReq()`) and the number of vblanks to wait.

```
Err WaitUntilVBL(Item ioreq, uint32 fields)
```

The task sleeps until the specified vblank count is reached.

When sending an `IOReq` manually, initialize the `IOInfo` structure as follows:

Example 10-4 *Initializing `IOInfo` for `TIMERCMD_DELAY_VBL`*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = TIMERCMD_DELAY_VBL;
ioInfo.ioi_Offset = numberOfVBlanksToWait;
```

Getting Repeated Notifications – Metronome Timing

Similar to the microsecond clock, the vblank clock can automatically send you signals at specified intervals – expressed in terms of vblanks.

The easiest way to do this is to create an `IOReq` and then call `StartMetronomeVBL()`, passing the `IOReq` item, the desired time interval in vblanks, and a signal mask indicating the signal that you want the timer to send to the calling task at the end of each time interval.

```
Err StartMetronomeVBL(Item ioreq, uint32 fields, int32 signal)
```

To stop the timer from sending these signals, call `StopMetronomeVBL()`, passing the `IOReq` item.

```
Err StopMetronomeVBL(Item ioreq)
```

Note that you could also start a metronome interval by creating and sending an `IOReq` with the `ioInfo.ioi_Command` field set to `TIMERCMD_METRONOME_VBL`.

Miscellaneous Timer Calls

The kernel provides additional functions that allow you to add, subtract, and compare time values. See the *3DO M2 Portfolio Programmer's Reference* for detailed descriptions of these functions.

- ◆ **AddTimes()** adds two time values together and gives you the sum.
- ◆ **Subtimes()** subtracts one time value from another and gives you the difference.
- ◆ **CompareTimes()** compares two time values to determine which came first.

- ◆ **TimeLaterThan()** returns a TRUE or FALSE value indicating whether one time value comes after another chronologically.
- ◆ **TimeLaterThanOrEqual()** returns a TRUE or FALSE value indicating whether one time value comes after or at the same time as another chronologically.

In addition, there are several calls that perform operations in terms of timer ticks (a hardware-dependent internal time representation): **AddTimerTicks()**, **SubTimerTicks()**, **CompareTimerTicks()**, **TimerTicksLaterThan()**, **TimerTicksLaterThanOrEqual()**, **ConvertTimerTicksToTimeVal()**, **ConvertTimeValToTimerTicks()**.

File Devices

When working with a 3DO M2 system, you will be working with several different kinds of 3DO file systems, such as

- ◆ A file system on a CD-ROM.
- ◆ A file system on an M2 microslot device used for saving game information.
- ◆ A file system in M2 ROM from which device drivers are loaded.
- ◆ A file system on the host Macintosh, which is mounted and seen as a 3DO M2 file system and used for development.

All the M2 file systems you are using at a particular time are mounted under the M2 root ("/") directory and can be listed by the "ls" M2 shell command.

When you open a file, an item of type `Device` is returned that serves as a handle to the file. This enables you to communicate with the filesystem that controls this file using the standard Portfolio I/O model.

A file device is only a pseudo-device, serving as a gateway to the underlying filesystem. When you send a command to a file device, the file system responsible for the file wakes up and executes the command.

Important Note About Microcard File Systems

Microcards "wear out" and begin to lose their ability to store data after about 10,000 write operations. To allow your users to get the maximum lifetime out of each microcard, you should structure your application to avoid unnecessary write operations.

Methods for Communicating With File Devices

As with other M2 devices, you can send a command to a file device by constructing and sending an `IOReq` (with its accompanying `IOInfo` structure) manually or by calling a function that constructs and sends the `IOReq` for you. Some operations can be done either way, others can only be done manually.

Functions are provided in the File folio and the FSUtils folio.

Here is a list of the ways that Portfolio provides for working with file systems. The list is arranged from lowest-level method to the highest-level method.

- ◆ Using the basic Portfolio I/O model to pass commands to the device via `IOReq`s with accompanying `IOInfo` structures.
This is the most basic method. You must use it to do certain kinds of operations, such as block-oriented I/O. See Chapter 9, “The Portfolio I/O Model” for a full description of the steps involved in this method.
- ◆ File functions in the File folio.
These functions perform certain basic file operations, such as creating and opening files, and some convenience operations, such as finding files.
- ◆ RawFile functions in the File folio.
These functions provide a byte-stream interface to files, similar to the one provided by C “stdio” routines, which is usually easier to use than the block-oriented interface.
- ◆ Directory functions in the File folio.
These functions let you manipulate and navigate directories.
- ◆ Utility functions in the FSUtils folio.
These functions let you perform higher-level utility operations, such as copying a whole directory tree.

This chapter only describes the operations for which you must construct and send an `IOReq` manually. In addition, some prerequisite File folio functions, such as `OpenFile()`, are also described. For the most part, however, File folio and FSUtils folio functions are described in Chapter 11, “The File System, File Folio, and FSUtils Folio”.

Creating, Deleting, Renaming, Getting Information about Files and Directories

Use the File folio functions `CreateFile()`, `DeleteFile()`, `CreateDirectory()`, `DeleteDirectory()`, `Rename()`, `ReadDir()`, `GetPath()` and other functions described in Chapter 11, “The File System, File Folio, and FSUtils Folio”.

Preparing to Send an IOReq to a File Device

First you open the file, using the File folio `OpenFile()` function. You provide a relative or absolute path name, and the function returns the device item for the file.

```
Item OpenFile(const char *path)
```

Then you create an `IOReq` and initialize the `IOInfo` structure, as described in Chapter 9, “The Portfolio I/O Model”. Use `CreateIOReq()` to create the `IOReq`, and fill in the `IOInfo` structure with name of the command to be sent to the device.

Getting Filesystem Status

After opening a file, you can obtain information about the filesystem the file resides on. To do this, send the `FILECMD_FSSTAT` command to the device. Initialize the `IOInfo` structure as follows:

Example 10-5 *Initializing IOInfo for FILECMD_FSSTAT*

```
IOInfo          ioInfo;
FileSystemStat  fsStat;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = FILECMD_FSSTAT;
ioInfo.ioi_Recv.iob_Buffer = &fsStat;
ioInfo.ioi_Recv.iob_Len  = sizeof(FileSystemStat);
```

Once the command returns successfully, you can look at the fields in the `fsStat` structure for the status information. The `fst_BitMap` field of the `FileSystemStat` structure indicates which fields in the rest of the structure are valid and can be examined. Different file systems cannot always provide all the information in a `FileSystemStat` structure. For example, if the `FSSTAT_SIZE` bit is set in `fst_BitMap`, it means the `fst_Size` field of the `FileSystemStat` structure is valid.

The fields in the `FileSystemStat` structure are:

- ◆ **fst_BitMap.** Indicates which fields of the structure contain valid data.
- ◆ **fst_RawDeviceName.** Indicates the name of the device the filesystem is connected to. For example, a filesystem that exists on a CD might be connected to the CD-ROM device driver, which in turn talks to the CD hardware.
- ◆ **fst_RawDeviceItem.** Indicates the item number of the device to which the filesystem is connected.
- ◆ **fst_RawOffset.** Indicates the offset within the device where the filesystem starts.
- ◆ **fst_MountName.** Indicates the name of the filesystem. This is the name you can use in `OpenFile()` statements to refer to the filesystem.

- ◆ **fst_CreateTime.** Indicates when the filesystem was created (first formatted).
- ◆ **fst_BlockSize.** Indicates the nominal size of data blocks in the filesystem. To determine the block size to use when reading and writing files, query the file's status and extract the block size from that information.
- ◆ **fst_Size.** Indicates the total size of the filesystem in blocks.
- ◆ **fst_MaxFileSize.** Indicates the maximum size of a file in the filesystem
- ◆ **fst_Free.** Indicates the total number of blocks currently not in use on the filesystem
- ◆ **fst_Used.** Indicates the total number of blocks currently in use on the filesystem.

Getting File Status

After opening a file, you can obtain information about the file by sending the CMD_STATUS command to the device. Initialize the IOInfo structure as follows:

Example 10-6 *Initializing IOInfo for CMD_STATUS*

```
IOInfo    ioInfo;
FileStatus fStat;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = CMD_STATUS;
ioInfo.ioi_Recv.iob_Buffer = &fStat;
ioInfo.ioi_Recv.iob_Len  = sizeof(FileStatus);
```

Once the command completes successfully, you can look at the fields in the fStat structure for the status information. Fields of interest are:

- ◆ **fs.ds_DeviceBlockSize.** The block size to use when reading or writing this file.
- ◆ **fs.ds_DeviceBlockCount.** The number of blocks of data in the file.
- ◆ **fs_ByteCount.** The number of bytes currently in the file.
- ◆ **fs_Type.** This field contains the four byte file type value that is associated with every file.
- ◆ **fs_Version.** This field specifies the version number of the file.
- ◆ **fs_Revision.** This field specifies the revision number of the file.
- ◆ **fs_Flags.** This field specifies various flags which describe certain attributes of the file. If FILE_IS_DIRECTORY is set in this field, it means that the device

actually refers to a directory. `FILE_IS_READONLY` specifies that the given device item currently cannot be written to. `FILE_USER_STORAGE_PLACE` is set to indicate that the device item refers to the root of a directory where it is possible to write user-level information (like saved games and such).

Allocating Blocks for a File

Before you can write data to a file, you must allocate enough free blocks in the file to hold the data to be written. This is done by sending the `FILECMD_ALLOCBLOCKS` command to the device. Initialize the `IOInfo` structure as follows:

Example 10-7 *Initializing IOInfo for FILECMD_ALLOCBLOCKS*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = FILECMD_ALLOCBLOCKS;
ioInfo.ioi_Offset = numBlocks;
```

The `numBlocks` variable contains the number of blocks by which to extend the file. If this value is negative, the size of the file is reduced by the specified number of blocks.

Writing Blocks of Data to a File

You must use the `CMD_BLOCKWRITE` command to do block-oriented writes to a file. All block-oriented write operations must be performed in full blocks. The size of the blocks can be obtained by sending a `CMD_STATUS` command to the file device. The write operation must also be aligned on a block boundary within the file. Initialize the `IOInfo` structure as follows:

Example 10-8 *Initializing IOInfo for CMD_BLOCKWRITE*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = CMD_BLOCKWRITE;
ioInfo.ioi_Offset = blockOffset;
ioInfo.ioi_Send.iob_Buffer = dataAddress;
ioInfo.ioi_Send.iob_Len = numBytes;
```

The value of the `blockOffset` field indicates the offset in blocks from the beginning of the file where the data is to be written. The `dataAddress` value points to the data that is to be written. Finally, the `numBytes` value indicates the number of bytes to write out. This value must be an even multiple of the block size for the file.

Marking the End of a File

Once you are done writing data to a file, you must mark the end of the file using the `FILECMD_SETEOF` command. This command tells the filesystem how many useful bytes of data are in the file. Because you can only transfer data in terms of blocks, sending this command tells the filesystem how many bytes of the last written block are useful bytes. Initialize the `IOInfo` structure as follows:

Example 10-9 *Initializing IOInfo for FILECMD_SETEOF*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = FILECMD_SETEOF;
ioInfo.ioi_Offset  = numBytesInFile;
```

Reading Blocks of Data From a File

Doing block-oriented reads from a file is done much the same way as block-oriented writes. You must supply the offset in blocks at which to start reading data, and the number of bytes of data to read. Initialize the `IOInfo` structure as follows:

Example 10-10 *Initializing IOInfo for CMD_BLOCKREAD*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = CMD_BLOCKREAD;
ioInfo.ioi_Offset       = blockOffset;
ioInfo.ioi_Recv.iob_Buffer = dataAddress;
ioInfo.ioi_Recv.iob_Len  = numBytes;
```

The `blockOffset` value indicates the offset in blocks from the beginning of the file from which data is read. `dataAddress` indicates an address in memory where the data will be copied once read from the filesystem. Finally, `numBytes` contains the number of bytes to read. This number must be an even multiple of the block size of the file.

Setting the File Type

Each file within a filesystem has a four-byte code associated with it. You can use the `FILECMD_SETTYPE` command to set this value.

Example 10-11 *Initializing IOInfo for FILECMD_SETTYPE*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = FILECMD_SETTYPE;
ioInfo.ioi_Offset       = fileType;
```

Setting the File Version and Revision

Each file within a filesystem has a one-byte version code and a one-byte revision code associated with it. You can use the `FILECMD_SETVERSION` command to set these values.

Example 10-12 *Initializing IOInfo for FILECMD_SETVERSION*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = FILECMD_SETVERSION;
ioInfo.ioi_Offset  = <version << 8> | revision;
```

Setting the Virtual Block Size

Each file within a filesystem has a virtual block size associated with it which may be different than the physical block size of the underlying device. You can set the virtual block size of a file only when that file is first created and contains no blocks. You can use the `FILECMD_SETBLOCKSIZE` command to set this value.

Example 10-13 *Initializing IOInfo for FILECMD_SETBLOCKSIZE*

```
IOInfo ioInfo;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command = FILECMD_SETBLOCKSIZE;
ioInfo.ioi_Offset  = blockSize;
```

Getting Directory Information

The `OpenFile()` function can be used to open directories. You can then use the `FILECMD_READDIR` command to scan the directory and obtain information about files it contains.

Example 10-14 *Initializing IOInfo for FILECMD_READDIR*

```
IOInfo          ioInfo;
DirectoryEntry  dirEntry;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = FILECMD_READDIR;
ioInfo.ioi_Offset       = fileNum;
ioInfo.ioi_Recv.iob_Buffer = &dirEntry;
ioInfo.ioi_Recv.iob_Len  = sizeof(DirectoryEntry);
```

The `fileNum` value indicates the number of the file within the directory to read. You start with file 1, and keep going until the I/O cannot be completed because there are no more files. The `dirEntry` structure will be filled out with information about the specified file.

Note that it is often more convenient to use the `OpenDirectory()` and `ReadDirectory()` functions provided by the File folio to read directory entries. However, to read the directory entry for a specific file, it is more efficient to send an `IOReq` with the `FILECMD_READENTRY` command, as shown below, because it does not require you to read through the directory entries for preceding files first, as the `ReadDirectory` command does.

Example 10-15 *Initializing IOInfo for FILECMD_READENTRY*

```
IOInfo          ioInfo;
DirectoryEntry  dirEntry;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = FILECMD_READENTRY;
ioInfo.ioi_Send.iob_Buffer = fileName;
ioInfo.ioi_Send.iob_Len  = strlen(fileName)+1;
ioInfo.ioi_Recv.iob_Buffer = &dirEntry;
ioInfo.ioi_Recv.iob_Len   = sizeof(DirectoryEntry);
```

Cleaning Up After I/O Operations

After completing I/O operations, you should delete the IOReq, using the Kernel folio `DeleteIOReq()` call, and then close the file, using the File folio `CloseFile()` function.

```
Err DeleteIOReq(Item ioreq)
int32 CloseFile(Item fileItem)
```

The `CloseFile()` function takes the file device item as an argument and returns an integer indicator of success or failure -- 0 or positive for success, negative for failure.

Serial Devices

A set of commands are provided to interact with serial ports allowing full control over standard serial capabilities.

Writing Data to a Serial Port

You must use the `CMD_STREAMWRITE` command to write data to a serial device. You can write an arbitrary number of bytes. The write operation is performed using the current serial configuration settings.

Example 10-16 *Initializing IOInfo for CMD_STREAMWRITE*

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command      = CMD_STREAMWRITE;
ioInfo.ioi_Send.iob_Buffer = dataAddress;
ioInfo.ioi_Send.iob_Len  = numBytes;
```

The dataAddress value points to the data that is to be written. The numBytes value indicates the number of bytes to write out. The command completes once all bytes have been written out.

Reading Data from a Serial Port

You must use the CMD_STREAMREAD command to read data from a serial device. You can read an arbitrary number of bytes. The read operation is performed using the current serial configuration settings. An optional time-out value can be specified to automatically suspend the read operation if data stops arriving for a certain time.

Example 10-17 *Initializing IOInfo for CMD_STREAMREAD*

```
IOInfo ioInfo;

memset(&ioInfo,0,sizeof(ioInfo));
ioInfo.ioi_Command      = CMD_STREAMREAD;
ioInfo.ioi_CmdOptions    = microSecondTimeOut;
ioInfo.ioi_Recv.iob_Buffer = dataAddress;
ioInfo.ioi_Recv.iob_Len  = numBytes;
```

The microSecondTimeOut value specifies the maximum number of microseconds allowed where no new data shows up at the serial port. When this number of microsecond passes, the I/O request is returned and io_Actual indicates the number of bytes actually read. If ioi_CmdOptions is set to 0, there is no time-out and the command will wait until the requested number of bytes has been read. The dataAddress value points to the location where the data read should be deposited. The numBytes value indicates the number of bytes to read.

Configuring the Serial Port

Serial ports support various features such as parity, variable bit width, stop bits, and so forth. You can use the `SER_CMD_SETCONFIG` command to change the configuration of a serial device. Changing the configuration discards any read or write operation currently in progress. It should therefore generally be done before establishing a serial connection.

Example 10-18 *Initializing IOInfo for SER_CMD_SETCONFIG*

```
IOInfo    ioInfo;
SerConfig config;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command    = SER_CMD_SETCONFIG;
ioInfo.ioi_Send.iob_Buffer = &config;
ioInfo.ioi_Send.iob_Len  = sizeof(config);
```

The `SerConfig` structure specifies the new configuration settings for the serial port. The fields of this structure are defined as:

- ◆ **sc_BaudRate.** Specifies the communication speed. The range of allowed speeds varies from system to system, but it can usually be as high as 64K baud.
- ◆ **sc_Handshake.** Specifies what kind of handshaking to perform. One of `SER_HANDSHAKE_NONE`, `SER_HANDSHAKE_SW`, or `SER_HANDSHAKE_HW`. No handshaking implies that there is no synchronization done at the driver level between the sender and the receiver to avoid overflows. Software handshaking is done by sending XON and XOFF bytes to start and stop data transmission. Finally, hardware handshaking involves using the RTS/CTS protocol to control data transmission.
- ◆ **sc_WordLength.** Specifies the length in bits of data transferred. This can go from `SER_WORDLENGTH_5` to `SER_WORDLENGTH_8`.
- ◆ **sc_Parity.** Specifies what kind of parity checking is desired. This can be either `SER_PARITY_NONE`, `SER_PARTIY_ODD`, `SER_PARITY_EVEN`, `SER_PARITY_MARK`, or `SER_PARITY_SPACE`.
- ◆ **sc_StopBits.** Specifies how many stop bits to use. This can be either `SER_STOPBITS_1` or `SER_STOPBITS_2`.
- ◆ **sc_OverflowBufferSize.** Specifies the size in bytes of the overflow buffer to allocate. The overflow buffer is used when there are no pending I/O requests to read data, yet more data is coming in from the serial port. With

properly structured client code, there is usually no need for an overflow buffer, so this value can be usually set to 0.

The default serial configuration is:

```
sc_BaudRate           = 57600
sc_Handshake          = SER_HANDSHAKE_NONE
sc_WordLength         = SER_WORDLENGTH_8
sc_Parity             = SER_PARITY_NONE
sc_StopBits           = SER_STOPBITS_1
sc_OverflowBufferSize = 0
```

Reading the Serial Port Configuration

You can query the current serial settings by using the `SER_CMD_GETCONFIG` command. You supply a pointer to a `SerConfig` structure which gets filled in with the current serial settings.

Example 10-19 *Initializing IOInfo for SER_CMD_GETCONFIG*

```
IOInfo  ioInfo;
SerConfig config;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = SER_CMD_GETCONFIG;
ioInfo.ioi_Recv.iob_Buffer = &config;
ioInfo.ioi_Recv.iob_Len  = sizeof(config);
```

Waiting for Particular Events

It is sometimes desirable to wait for certain state transitions to occur on the serial port. You can use the `SER_CMD_WAITEVENT` command to specify events to wait for. Whenever any of these events occur, the I/O request is returned to you with an indication of exactly which events have occurred.

A bitmask is provided in the `ioi_CmdOptions` field to specify which events to wait for. When any of these events occur, the I/O request is returned and the `io_Info.ioi_CmdOptions` field is set to a mask of the events that occurred. The supported events that can be specified in `ioi_CmdOptions` are:

- ◆ **SER_EVENT_CTS_SET.** Waits for the CTS line to go high.
- ◆ **SER_EVENT_DSR_SET.** Waits for the DSR line to go high.
- ◆ **SER_EVENT_DCD_SET.** Waits for the DCD line to go high.

- ◆ **SER_EVENT_RING_SET.** Waits for the RING indicator to be set.
- ◆ **SER_EVENT_CTS_CLEAR.** Waits for the CTS line to go low.
- ◆ **SER_EVENT_DSR_CLEAR.** Waits for the DSR line to go low.
- ◆ **SER_EVENT_DCD_CLEAR.** Waits for the DCD line to go low.
- ◆ **SER_EVENT_RING_CLEAR.** Waits for the ring indicator to be cleared.
- ◆ **SER_EVENT_OVERFLOW_ERROR.** Waits for an overflow error to occur.
- ◆ **SER_EVENT_PARITY_ERROR.** Waits for a parity error to occur.
- ◆ **SER_EVENT_FRAMING_ERROR.** Waits for a framing error to occur.
- ◆ **SER_EVENT_BREAK.** Waits for a break signal to be received.

It's useful to issue an I/O request that waits for error events to occur. This will tell you automatically when errors occur on the serial line.

Controlling Serial Lines

You can control the state of the DTR and RTS serial lines directly by using the **SER_CMD_SETDTR** and **SER_CMD_SETRTS** commands respectively. For both commands, you set **ioi_CmdOptions** to 1 to set the line high, and 0 to set the line low.

If the serial port is currently set for hardware handshaking, then the **SER_CMD_SETRTS** command cannot be used, as the state of the RTS line is controlled by the driver directly to throttle data flow when needed.

Sending Break Signals

You can send a serial break signal by using the **SER_CMD_BREAK** command. You can set the **ioi_CmdOptions** field to specify the length of the break signal in microseconds. If this value is 0, a default break duration is assumed.

Getting Serial State

You can obtain the current state of the serial port by using the **SER_CMD_STATUS** command. You supply a pointer to a **SerStatus** structure which gets filled in by the serial driver to reflect the current serial state. Information in the **SerStatus** structure includes the number of errors that have occurred, the number of bytes that have been dropped due to overflows, and a bitmask indicating the state of various serial bits include RTS, DTR, etc.

Example 10-20 *Initializing IOInfo for SER_CMD_STATUS*

```
IOInfo    ioInfo;
SerStatus status;

memset(&ioInfo, 0, sizeof(ioInfo));
ioInfo.ioi_Command      = SER_CMD_STATUS;
ioInfo.ioi_Recv.iob_Buffer = &status;
ioInfo.ioi_Recv.iob_Len  = sizeof(status);
```

The File System, File Folio, and FSUtils Folio

This chapter describes the 3DO file system. It includes sample code that illustrates how to use File folio calls.

This chapter contains the following topics:

Topic	Page Number
The 3DO File System Interface	137
File Functions	141
RawFile Functions	145
Directory Functions	151
File Folio Examples	154
The FSUtils Folio - File System Utilities	165

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

The 3DO File System Interface

The 3DO file system is, in essence, a high-level device driver that interposes itself between an application and the "raw" disk device. The application issues a call to open a specific disk file and in return receives an item for a device node. This device item (referred to internally as an "open file") can be treated like most other device items—you can create one or more IOReq items for the device item, and send these IOReq items to the device to read data from the file.

The file system architecture is implemented as a combination of the following elements:

- ◆ The **File folio**, which provides high-level functions to interface to the lower-level components.
- ◆ The **driver**, which manages the file system and open file devices.
- ◆ The **daemon** task, manages I/O scheduling.

Caution: Remember to avoid unnecessary write operations to microcards. See "Important Note About Microcard File Systems" on page 123.

Files and File Functions

Files are collections of blocks, each block containing a fixed number of bytes. The block size of a file is determined generally by the block or sector size of the device on which it resides. For example, on a CD-ROM file system, most files have a block size of 2048 bytes.

The file system deals with files exclusively in terms of blocks. However, it does maintain a logical size in bytes for each file. This logical size indicates the number of bytes actually used within the blocks allocated for the file.

The File functions in the File folio open and close files for block-oriented operations and, in addition, perform general operations, such as creating, deleting, and renaming files.

RawFiles and RawFile Functions

The File folio's RawFile functions treat a file as a stream of bytes. Using these functions, you can read, write, and seek in files in a manner that is similar to the C "stdio" routines.

Directories and Directory Functions

The 3DO file system implements directories as ordinary disk files that are flagged as a specific type. The File folio's Directory functions create, delete, read, manipulate, and navigate directories.

You can also use the `OpenFile()` function to open a directory as a normal file and then send commands to it using the `IOReq` mechanism described in Chapter 10 in the section entitled "File Devices" on page 123.

Normally, you should use the directory functions because they are easier, but there is one circumstance, reading an entry for a particular file, where you would use an `IOReq`. Note that, in M2, the directory functions are supported for all file systems.

Avatars

The 3DO CD-ROM file system supports the unique concept of avatars. Each file consists of one or more avatars, placed in various locations across the disk. Each avatar is an exact, coequal image of the data in the file.

How Avatars Work

The file driver keeps track of the current position of the device's read head. When asked to read data, it automatically chooses the "closest avatar in good condition" of each block of data it is asked to read. If it must perform more than one read, the driver sorts the read requests to minimize the amount of head-seeking. It honors the I/O request priorities during this process—completing all requests of one priority before scheduling any requests of a lower priority.

If the device has trouble reading a data block contained within one avatar of a file, the file driver marks that avatar as flawed and reissues the read request—thus searching out another avatar on the disk, if one exists.

It is not necessary, nor is it desirable, for all files in a 3DO file system to have the same number of avatars. Critical directories, or files accessed frequently throughout the execution of the application, can have as many as a dozen avatars scattered across the disk. Large files, or those containing noncritical data, may have only one avatar.

Placement of a File's Avatars

The placement of a file's avatars can have a great effect on the performance of a Portfolio game application. If your application opens and reads several files in sequence, and the avatars lie some distance away from one another on the CD-ROM, the CD-ROM drive must *seek* from one file to another; your program must wait while it does so. However, if the avatars are located close together, ideally, right next to one another, the CD-ROM drive needs to seek less, and perhaps not at all. Every seek on a CD-ROM drive takes tens to hundreds of milliseconds; it's important to minimize the number of seeks for your application to perform well.

The software tool that lays out a Portfolio CD-ROM can optimize the avatar locations on your files in an effective, semiautomatic fashion. It can create two or more avatars of frequently used files, to reduce the distance needed to seek to the file. It can even weave portions of different files together into a single accelerated-access file called *Catapult*, eliminating most of the seeks that occur during the system start-up process.

The layout optimization and multiple-avatar speedups, and the Catapult acceleration, are completely transparent to your application. You do not need to change any application code to take advantage of them. They do not change the functionality of the file system; they simply improve its performance.

Instructions on doing an optimized CD-ROM layout with Catapult acceleration can be found in the *3DO CD-ROM Mastering Guide*. You should do an optimized and Catapult-accelerated layout of your application before you submit it for encryption. If you submit a non-optimized CD-ROM image for encryption, the company may ask you to optimize and resubmit it.

Pathnames and Filenames

To access files and directories in 3DO file systems, you use pathnames. A pathname describes the location of a file within the file system hierarchy. 3DO file systems use pathnames made up of components separated by slashes ("/") similar to the pathnames used in UNIX.

Like UNIX, 3DO file systems allow absolute or relative pathnames. An absolute pathname always starts with a slash ("/") and describes the file's location with respect to the top of the file system hierarchy.

A relative pathname does not begin with a slash. Normally a relative pathname describes a file's location relative to the current directory. Each task in Portfolio has a current directory associated with it. A relative pathname can be in one of three forms:

```
filename dir/filename dir1/dir2/(more)/dirN/filename
```

Some commands allow you to specify a relative pathname in relation to a directory other than the task's current directory.

For example:

- ◆ In the `OpenFile()` call, pathnames specify a path relative to the current task's current directory.
- ◆ In the `OpenFileInDir()` call, pathnames specify a path relative to a directory whose item is specified in the call.

In addition, three special conventions are used to further describe the 3DO directory structure. A path component of a period (".") indicates the current directory. A path component of two periods ("..") indicates the parent directory. Finally, a caret ("^") indicates the root of the file system.

Each pathname component should not be longer than 31 characters. In addition, certain characters are not allowed in component names. These illegal characters are slash ("/"), dollar sign ("\$"), left bracket ("{"), right bracket ("}"), and a pipe ("|").

File Functions

Creating and Deleting Files

To create a new file, use the `CreateFile()` function, which takes a relative (to the current directory) or absolute pathname as an argument and returns a code indicating success (`code>=0`) or failure (`code<0`).

```
Err CreateFile(const char *path)
```

You can also use `CreateFileInDir()`, which performs the same function but takes both a directory item and a pathname as arguments. If the specified pathname is a relative path, it is considered relative to the directory whose item was specified rather than relative to the current directory.

```
Err CreateFileInDir(Item dirItem, const char *path)
```

To delete a file, use the `DeleteFile()` function, which takes a relative or absolute pathname as an argument and returns a code indicating success (`code>=0`) or failure (`code<0`).

```
Err DeleteFile(const char *path)
```

You can also use `DeleteFileInDir()`, which works in a similar way as `CreateFileInDir()`.

```
Err DeleteFileInDir(Item dirItem, const char *path)
```

Renaming Files and Directories

To rename a file or directory, use the `Rename()` function. You pass two arguments: the pathname (relative or absolute) of the file or directory you want to rename and the desired new filename (i.e., last element of the pathname).

The filename of the target file or directory is changed to the new filename. Note that the renamed file or directory remains within the same parent directory. Only the last element of its pathname changes.

The `Rename()` function returns a code indicating success (`code>=0`) or failure (`code<0`).

```
Err Rename(const char *path, const char *newName)
```

Opening and Closing Files (For Block-Oriented Operations)

To open a file for block operations of the type described in Chapter 10 “Portfolio Devices”, use the `OpenFile()` function. This function takes a relative or absolute pathname as an argument and returns the item for the opened file. A negative return code indicates failure.

```
Item OpenFile(const char *path)
```

You can also use the `OpenFileInDir()` function, which will consider a specified relative path as being relative to the directory whose item is specified.

```
Item OpenFileInDir(Item dirItem, const char *path)
```

To close a file that was opened for block-oriented operations, use the `CloseFile()` function, which takes a relative or absolute pathname and returns a code indicating success (`code>=0`) or failure (`code=0`).

```
int32 CloseFile(Item fileItem)
```

Setting File Attributes

To change selected attributes of a file, use the `SetFileAttrs()` function. This function takes as arguments the relative or absolute pathname of the file and a pointer to a tag array. You can use this function to set the file's 4-byte id (use tag `FILEATTRS_TAG_FILETYPE`), version number (use tag `FILEATTRS_TAG_VERSION`), revision number (use tag `FILEATTRS_TAG_REVISION`), or blocksize (use tag `FILEATTRS_TAG_BLOCKSIZE`).

```
Err SetFileAttrs(const char *path, const TagArg *tags)
```

Finding Files

The `FindFileAndIdentify()` and `FindFileAndOpen()` functions let you find a file by specifying a partial pathname along with version and revision number search criteria. The most common use of these functions is to find the most recent version of an operating system routine when the M2 ROM contains one version and the CD-ROM contains another version.

The `FindFileAndIdentify()` function finds the file and puts the full absolute pathname into a buffer that you provide.

```
Err FindFileAndIdentify(char *pathBuf, int32 pathBufLen,  
                        const char *partialPath, TagArg *tags)
```

The `FindFileAndOpen()` function finds the file in the same way but then opens the file and returns its item rather than just identifying it.

```
Item FindFileAndOpen(const char *partialPath, TagArg *tags)
```

With both of these functions, you must specify a relative partial pathname for the file to look for and a set of tag arguments indicating how and where to search. There are three groups of tag arguments.

- ◆ Tag arguments that indicate the version and revision number to look for.
If not specified, the default is just to search for files whose pathname matches the one you specified.

- ◆ Tag arguments that indicate which file to select if multiple files meet the search criteria – i.e.,
 - ◆ The first file that matches the search criteria or
 - ◆ The file with the highest version and revision number that matches the search criteria.

If not specified, the default is “highest match”. Thus, if you specified no version or revision number and no “first” or “highest” tag, the function would return the file with the highest version and revision number whose pathname matched the one specified.

- ◆ Tag arguments that indicate which file systems or specific directories to search in.

When the function executes, the tags are processed in the order that they are encountered as follows:

1. The function stores any arguments indicating version/revision number and how to deal with multiple hits.
2. When the function encounters the first tag indicating a directory or file systems to search, it searches the specified directory or set of file systems using the search criteria provided so far.

If a match is found, and the tag indicating “find the first match” was specified, the function stops and returns with the found pathname.

If a match is found, and the tag indicating “find the highest match” was specified, the function saves the found pathname and continues looking for the highest match in that directory or set of filesystems.

If no match is found, the function continues.

3. If more tags containing search criteria are encountered, they are added to the stored search criteria, overriding previous criteria when appropriate.
4. When the function encounters the next tag indicating a directory or filesystems to search, it searches the specified directory or set of filesystems using the search criteria provided so far.

If a match is found and the function is searching for the highest match, the function compares the version/revision of the found pathname to the highest match found in previously searched directories and filesystems. It then saves the match with the highest matching version/revision and continues.

5. When the function has searched all the specified directories and file systems, it completes and returns with a single found pathname or with an error code.

The following sections describe the tags in more detail.

Tag Arguments Specifying Version and Revision Number Search Criteria

- ◆ To find a file whose version number is equal to a particular version number, specify the tag `FILESEARCH_TAG_VERSION_EQ (uint32 v)`

where "v" is the version number -- e.g.,

```
FILESEARCH_TAG_VERSION_EQ, (TagData) 5
```

(Note that `TagData` is the typedef used to cast tag arguments. For an explanation of how tag arguments are specified, see Chapter 3 "Using Tag Arguments".)

The following tags can be specified in a similar manner to find a file whose version number is not equal to, less than, greater than, less than or equal to, or greater than or equal to a particular version number:

```
FILESEARCH_TAG_VERSION_NE, FILESEARCH_TAG_VERSION_LT,  
FILESEARCH_TAG_VERSION_GT, FILESEARCH_TAG_VERSION_LE,  
FILESEARCH_TAG_VERSION_GE.
```

- ◆ To find a file whose revision number is equal to a particular revision number, specify the tag `FILESEARCH_TAG_REVISION_EQ (uint32 r)`

where "r" is the revision number -- e.g.,

```
FILESEARCH_TAG_REVISION_EQ, (TagData) 3
```

The following tags can be specified in a similar manner to find a file whose revision number is not equal to, less than, greater than, less than or equal to, or greater than or equal to a particular revision number:

```
FILESEARCH_TAG_REVISION_NE, FILESEARCH_TAG_REVISION_LT,  
FILESEARCH_TAG_REVISION_GT, FILESEARCH_TAG_REVISION_LE,  
FILESEARCH_TAG_REVISION_GE.
```

- ◆ To find a file whose combined version and revision number is equal to a particular combined version and revision number, specify the tag `FILESEARCH_TAG_REVISION_EQ (uint32 0x0v0r)`

where "0x0v0r" is the version and revision number in hex -- e.g.,

```
FILESEARCH_TAG_REVISION_EQ, (TagData) 0x0503
```

The following tags can be specified in a similar manner to find a file whose combined version and revision number is not equal to, less than, greater than, less than or equal to, or greater than or equal to a particular version and revision number: `FILESEARCH_TAG_VERREV_NE`,

```
FILESEARCH_TAG_VERREV_LT, FILESEARCH_TAG_VERREV_GT,  
FILESEARCH_TAG_VERREV_LE, FILESEARCH_TAG_VERREV_GE.
```

- ◆ To indicate that files with no version and revision number should be considered Version 0, Revision 0, specify the tag

```
FILESEARCH_TAG_NOVERSION_IS_0_0 (0)
```

This is the default.

- ◆ To indicate that files with no version and revision indicator should be excluded from the search completely, specify the tag

`FILESEARCH_TAG_NOVERSION_IGNORED (0)`

Tag Arguments Indicating How to Resolve Multiple Hits

- ◆ To select the first file that matches the search criteria, specify the tag

`FILESEARCH_TAG_FIND_FIRST_MATCH (0)`

- ◆ To select the file with the highest search value (e.g., version/revision number) that matches the search criteria, specify the tag

`FILESEARCH_TAG_FIND_HIGHEST_MATCH (0)`

This is the default.

Tag Arguments Indicating File Systems or Directories to Search

- ◆ To search all file systems, specify the tag

`FILESEARCH_TAG_SEARCH_FILESYSTEMS (uint32 flags)`

where the following exclusionary option flags can be set singly or combined by OR'ing them together:

- ◆ `DONT_SEARCH_QUIESCENT` means do not search any file systems that are quiescent.
- ◆ `DONT_SEARCH_NO_CODE` means do not search any file systems whose driver module has not been loaded.
- ◆ `DONT_SEARCH_USER_STORAGE` means do not search any file systems whose root directory has the "This to store user files" flag set.
- ◆ `DONT_SEARCH_UNBLESSSED` means do not search any file systems that are not labelled as having 3DO software in them.

Example of specifying two flags OR'd together:

```
..., (TagData) (DONT_SEARCH_QUIESCENT | DONT_SEARCH_UNBLESSSED)
```

- ◆ To search in a specific directory, specify the tag

`FILESEARCH_TAG_SEARCH_PATH (char *)`

with the pathname (relative or absolute) of the directory in which to search.

- ◆ To search in a specific directory or file system that has already been opened, specify the tag `FILESEARCH_TAG_ITEM (Item)`

with the `Item` of the open directory or file system in which to search.

RawFile Functions

The file folio provides a collection of routines to perform traditional byte-oriented I/O operations on files. The various RawFile functions let you do efficient byte-oriented reads and writes, and to get and set file attributes. They are called

RawFile functions in relation to the fact that there is no sophisticated buffering scheme in these functions. They are simple and efficient interfaces to the block-oriented file system interface, which provide a high throughput for operations typically performed on a 3DO system.

The RawFile functions operate using the traditional byte stream I/O model. A file must be opened before use, and closed when no longer needed. An opened file has a file cursor, which indicates the current position within the file. When reading data from a file, the read operation always starts from the current cursor position onward. Writing data also starts writing at the cursor position. Reads and writes automatically advance the cursor in the file. It is also possible to manually move the file cursor, known as seeking.

Whenever any operation on a file fails, attempts at doing any additional operations will also fail and continue to return the same error code. When you close the file using `CloseRawFile()`, this error code will also be returned to you. This method of operation is very convenient, as you can do all the reads and writes of a file without checking error codes, and only look at the return value of `CloseRawFile()` to determine if a failure occurred.

Opening a RawFile

To obtain a RawFile, you use the `OpenRawFile()` function:

```
Err OpenRawFile(RawFile **file, const char *path,
                FileOpenModes mode);
```

The first parameter is a pointer to a variable of type `RawFile`. This is where the file folio will put a handle to the `RawFile` structure, which you later use to access the file contents.

The second parameter is the name of the file to open. This string follows the same format as the pathnames accepted by the `OpenFile()` function. The pathname can therefore be relative or absolute, and can contain file aliases.

The third parameter specifies the mode in which the file should be opened. The mode used determines which operations are allowed on the file. It also helps the file folio decide what is the best way to deal with this file internally to optimize resource usage and maximize performance. The possible file open modes are:

- ◆ **FILEOPEN_READ**. This opens the file for reading only. Attempts to write to the file will fail. If the file doesn't exist when `OpenRawFile()` is called, the open operation will fail and return an error code.
- ◆ **FILEOPEN_WRITE**. This opens the file for writing only. Attempts to read from the file will fail. If the file doesn't exist when `OpenRawFile()` is called, the file will be created automatically. If the file does already exist, the contents will remain intact, and the file cursor will be put at the beginning of the file.

- ◆ **FILEOPEN_WRITE_NEW.** This opens the file for writing only. Attempts to read from the file will fail. If the file doesn't exist when `OpenRawFile()` is called, the file will be created automatically. If the file does exist, the old contents are erased and the file is reinitialized to be like a brand new file.
- ◆ **FILEOPEN_WRITE_EXISTING.** This is the same as `FILEOPEN_WRITE` except that if the file does not exist, the call will fail.
- ◆ **FILEOPEN_READWRITE.** This is similar to `FILEOPEN_WRITE` except that the file can also be read.
- ◆ **FILEOPEN_READWRITE_NEW.** This is similar to `FILEOPEN_WRITE_NEW` except that the file can also be read.
- ◆ **FILEOPEN_READWRITE_EXISTING.** This is similar to `FILEOPEN_WRITE_EXISTING` except that the file can also be read.

`OpenRawFile()` returns ≥ 0 if the file was opened, or a negative error code for failure. When the open succeeded, the variable pointed to by the first argument to the function will contain the `RawFile` handle. You pass this value to other `RawFile` functions to perform operations on the opened file.

You can also use the `OpenRawFileInDir()` function, which performs the same operation but treats a target relative path name as being relative to the indicated directory rather than to the current directory.

Reading From a RawFile

To read from an opened `RawFile`, you use `ReadRawFile()`:

```
int32 ReadRawFile(RawFile *file, void *buffer, int32 numBytes);
```

The first argument is a `RawFile` handle, as obtained from `OpenRawFile()`. The second argument is a pointer to a writable memory buffer where the data read from the file will be put. The third argument specifies the number of bytes of data to read. The supplied memory buffer must be large enough to accommodate that number of bytes.

The function returns the total number of bytes read from the file. If you asked to read more data than there was in the file, the function will read as many bytes as possible. A negative error code is returned if the read operation failed. In such a case, nothing can be presumed about the contents of the supplied data buffer.

Writing to a RawFile

To write data to an opened `RawFile`, you use `WriteRawFile()`:

```
int32 WriteRawFile(RawFile *file, const void *buffer, int32 numBytes);
```


The first argument is a `RawFile` handle, as obtained from `OpenRawFile()`. The second argument is a pointer to a memory buffer where the data to be written is located. The third argument specifies the number of bytes of data to write. The supplied memory buffer should be at least this number of bytes large.

The function returns the total number of bytes written to the file. A negative error code is returned if the write operation failed. In such a case, nothing can be presumed about the contents of the file, and it should probably be deleted.

Writing at the end of a file automatically extends the size of the file.

Setting the Size of a RawFile

To preallocate space for a file or to change the amount of space already preallocated, use the `SetRawFileSize()` function. The most common reason for doing this is to determine ahead of time if there is enough space in the target file system to write or expand your file. This function is executed against a target file that has already been created and opened.

```
Err SetRawFileSize(RawFile *file, uint32 newSize)
```

You pass as arguments a pointer to the open file and a count of the number of bytes that you expect the file to occupy. If that count is greater than the current allocation, additional blocks are allocated. The blocks are not necessarily contiguous. If the count is less, then blocks are deallocated. If the file cursor is now pointing beyond the new end of file, it is moved back.

You do not have to preallocate space for a file. The system creates a file with a default space allocation of zero and will automatically expand the file to accommodate new data as long as there is free space in the file system.

If you do preallocate space, the preallocation does not restrict the amount of space that the file can occupy. A file can continue to expand beyond its preallocated space as long as there is enough free space in the file system.

Seeking in a RawFile

To change the file cursor position in an opened `RawFile`, you use `SeekRawFile()`:

```
int32 SeekRawFile(RawFile *file, int32 position, FileSeekModes mode);
```

The first argument is a `RawFile` handle, as obtained from `OpenRawFile()`. The second argument specifies the position to which to move the file cursor. The meaning of this argument depends on the value of the seek mode argument. The available seek modes are:

- ◆ **FILESEEK_START.** Indicates that the supplied position is relative to the beginning of the file. So a position of 10 would put the file cursor at byte #10 (the eleventh byte) within the file.
- ◆ **FILESEEK_CURRENT.** Indicates that the supplied position is relative to the current position within the file. A positive position value moves the cursor forward in the file by that many bytes, while a negative value moves the cursor back by that number of bytes.
- ◆ **FILESEEK_END.** Indicates that the supplied position is relative to the end of the file. The position value should be a negative value.

You are never allowed to move the file cursor beyond the current bounds of the file. Attempting to do so will simply leave the cursor at the farthest location it could go.

The function returns the previous file cursor position within the file, or a negative error code if the seek operation failed. When a seek fails, the position of the file cursor is assumed not to have changed.

Clearing Error Conditions for an Open RawFile

You can clear error conditions in an open RawFile by calling `ClearRawFileError()`:

```
Err ClearRawFileError( RawFile *file );
```

You pass a RawFile handle, as obtained from `OpenRawFile()`.

When an error occurs reading, writing, or seeking in an opened file, an error state indicator is set that prevents further operations against that file. All further operations against that file are rejected, returning error codes.

`ClearRawFileError()` resets the error state indicator and allows file operations against that file to continue.

You would use `ClearRawFileError()` in cases where the error is correctable and you want to continue. For example, if the error happened because the user removed the media, you might want to prompt the user to reinsert the media, clear the error state, and then continue file operations once the media had been successfully remounted.

Getting Information About a RawFile

You can get information about an opened RawFile by calling `GetRawFileInfo()`:

```
Err GetRawFileInfo(const RawFile *file, FileInfo *info, uint32  
infoSize);
```

The first argument is a `RawFile` handle, as obtained from `OpenRawFile()`. The second argument specifies a pointer to a `FileInfo` structure that will receive the information. The third argument is always set to the size of the `FileInfo` structure, namely `sizeof(FileInfo)`.

The fields of the `FileInfo` structure have the following meaning:

- ◆ **fi_File**. This is the item number of the `Device` item used to interact with this file. You can use this item to send custom I/O requests to the file directly.
- ◆ **fi_FileType**. Every file in a 3DO file system has a 4-byte code associated with it. This value is used by the user-interface software to glean high-level information about the contents of the file without having to read it. This field returns the 4-byte code for the file.
- ◆ **fi_ByteCount**. Specifies the current number of bytes in the file.
- ◆ **fi_BlockCount**. Specifies the current number of blocks in the file.
- ◆ **fi_BlockSize**. Specifies the block size being used for this file.
- ◆ **fi_BytePosition**. Specifies the current cursor position within the file, relative to the beginning of the file.
- ◆ **fi_Version**. Specifies the version of the file, as stored within the file system.
- ◆ **fi_Revision**. Specifies the revision of the file, as stored within the file system.

The function returns ≥ 0 for success, or a negative error code for failure. When a failure occurs, the contents of the supplied `FileInfo` structure are undefined.

Setting the Attributes of a RawFile

You can set various attributes for an opened `RawFile` by calling `SetRawFileAttrs()`:

```
Err SetRawFileAttrs(RawFile *file, const TagArg *tags);
```

The first argument is a `RawFile` handle, as obtained from `OpenRawFile()`. The following argument is a standard tag list, that can contain the following tags:

- ◆ **FILEATTRS_TAG_FILETYPE** (`PackedID`). This tag lets you specify the 4-byte file type that is associated with every file, and is used by user-interface code to identify the contents of a file.
- ◆ **FILEATTRS_TAG_VERSION** (`uint8`). Lets you specify the version associated with the file.
- ◆ **FILEATTRS_TAG_REVISION** (`uint8`). Lets you specify the revision associated with the file.
- ◆ **FILEATTRS_TAG_BLOCKSIZE** (`uint32`). Lets you specify the logical block size to use for this file. This tag can only be used if the file is currently empty.

The function returns ≥ 0 if all the attributes were set, or a negative failure code if at least one attribute could not be set.

Closing a RawFile

When you are done using a RawFile, you should close it using `CloseRawFile()`:

```
Err CloseRawFile(RawFile *file);
```

The only argument is a RawFile handle, as obtained from `OpenRawFile()`. This value may actually be NULL and this function will simply ignore it.

The return value for this function will be ≥ 0 if all I/O operations to this file were successful. A negative return indicates that a failure occurred, and the error code will correspond to the error code generated by the I/O operation that failed.

Once a file is closed, it can no longer be accessed, until it is opened again.

Directory Functions

The File folio provides functions that allow you to create, delete, and get information about directories. These functions also let you get information about currently mounted file systems.

Creating and Deleting Directories

To create a new directory, use the `CreateDirectory()` function. You specify the relative or absolute path of the directory to create. The function creates the directory or returns a negative error code.

```
Err CreateDirectory(const char *path)
```

You can also use the `CreateDirectoryInDir()` function, which performs the same operation but treats a target relative path name as being relative to the indicated directory rather than to the current working directory.

To delete an existing directory, use the `DeleteDirectory()` function. You specify the relative or absolute path of the directory to delete. The function deletes the directory from the file system or returns a negative error code.

```
Err CreateDirectory(const char *path)
```

You can also use the `DeleteDirectoryInDir()` function, which performs the same operation but treats a target relative path name as being relative to the indicated directory rather than to the current directory.

Opening and Closing Directories

You can open a directory to read by specifying the directory's pathname or the directory's item number.

`OpenDirectoryPath()` opens a directory by specifying its pathname.

```
Directory *OpenDirectoryPath( const char *thePath )
```

This function opens a directory, allocates a new `Directory` structure, and prepares for a traversal of the contents of the directory. It returns a pointer to a `Directory` structure or `NULL` if an error occurs.

`OpenDirectoryItem()` opens a directory by referencing its item number.

```
Directory *OpenDirectoryItem( Item openFileItem )
```

It allocates a new `Directory` structure, opens the directory, and prepares for a traversal of the contents of the directory. `OpenDirectoryItem()` returns a pointer to the `Directory` structure, or `NULL` if an error occurs. Typically, you obtain the item number of a directory by calling `OpenFile()` on the directory, or by calling `GetDirectory()`.

`CloseDirectory()` closes a directory that was previously opened using `OpenDirectoryItem()` or `OpenDirectoryPath()`.

```
void CloseDirectory( Directory *dir )
```

All resources get released.

Finding and Changing the Current Directory

Each task in the Portfolio environment has a current directory associated with it. The current directory is the starting location for relative pathnames that are used with various File folio calls that take a pathname as an argument.

When an application is loaded from a file, its current directory is set to the location of the program. This makes it easier for the new task to find any files it needs.

The File folio provides two calls, `GetDirectory()` and `ChangeDirectory()`, to determine and change the location of the current directory of the current task.

`GetDirectory()` returns the item number of the current directory of the calling task.

```
Item GetDirectory( char *pathBuf, int32 pathBufLen )
```

If `pathBuf` is non-`NULL`, it must point to a buffer of writable memory whose length is given in `pathBufLen`; the absolute pathname of the current working directory is stored into this buffer.

`ChangeDirectory()` changes the current directory of the current task to the absolute or relative location specified by the path.

```
Item ChangeDirectory( const char *path )
```

The function returns the item number of the new directory, or a negative error code if an error occurs.

Reading a Directory

`ReadDirectory()` reads the next entry from the specified directory.

```
Err ReadDirectory( Directory *dir, DirectoryEntry *de )
```

This function gets information about the next directory entry, and deposits this information in the supplied `DirectoryEntry` structure. The contents of this structure can then be examined to get details about the entry. The function returns an error code if all entries in the directory have been processed.

The following are some of the important fields in the directory entry:

- ◆ **de_FileName.** Specifies the name of the file described by the entry.
- ◆ **de_Flags.** Contains flags describing characteristics of the file. For example, `FILE_IS_DIRECTORY`, indicates that the file is a nested directory, and `FILE_IS_READONLY` indicates that the file can be read but not written to.
- ◆ **de_BlockSize.** Specifies the size, in bytes, of each block allocated for the file.
- ◆ **de_ByteCount.** Specifies the logical count of the number of useful bytes within the blocks allocated for the file.
- ◆ **de_BlockCount.** Specifies the number of blocks allocated for the file.

Note that to read the entry for a specific file, you would have to open the directory and read the entries for preceding files first to get to the one you want. However, you can read a specific entry by opening the directory as a normal file and sending an `IOReq` with the `FILECMD_READENTRY` command, as documented in Chapter 10 in the section entitled “Getting Directory Information” on page 130.

Mounting and Dismounting File Systems

Normally, you do not have to mount and dismount file systems. The M2 system mounts and dismounts filesystems at the appropriate times, such as when media are inserted or removed. On the rare occasions that you do need to mount and dismount file systems, use the `MountFilesystem()` and `DismountFilesystem()` functions.

Minimizing File Systems

You may periodically want to have the system minimize the amount of memory occupied by file systems not currently in use – without actually dismounting them. To do this, use the `MinimizeFilesystem()` function, passing it a flag field indicating the minimization operations you want to perform.

```
void MinimizeFilesystem(uint32 minimizationFlags)
```

In most cases, you will want to set both the `FSMINIMIZE QUIESCE` and `FSMINIMIZE UNLOAD` flags. This will quiesce unused file systems completely but leave them in the list of available file systems. A quiesced file system is reactivated automatically if an open request is issued against any file in it.

- ◆ `FSMINIMIZE QUIESCE` - Quiesce file systems: i.e., flush unused file structures, close metadata files, and reduce per-file system RAM usage to the bare essentials. A file system cannot become quiescent if there are any files or directories open on it or if any task's current directory is in it. A quiescent file system will automatically be reactivated if you try to open any file in it.
- ◆ `FSMINIMIZE UNLOAD` - Quiesce the file systems, and then unload the filesystem interpreter code from memory where possible. The code will be reloaded automatically when the filesystem is opened again.
- ◆ `FSMINIMIZE DISMOUNT` - Quiesce the file systems, unload the code, and completely dismount the file systems. This option is dangerous and not often used because you may unmount a file system required to run your system.

Finding Mounted File Systems

You can obtain a list of the mounted file systems by scanning the `"/"` directory. If you open this directory and call `ReadDirectory()` on it, you can then iterate through all of the currently mounted file systems. The `DirectoryEntry` structure returned by `ReadDirectory()` for each file system will give you information about the file system, such as whether the file system is read-only (i.e., `de_Flags FILE_IS_READONLY` bit is set).

File Folio Examples

This section provides code listings of programs that use the File folio and the file system.

Displaying Contents of a File

Example 11-1 is a program listing that demonstrates how to read a file using the byte stream calls, and output its contents to the debugging terminal.

Example 11-1 *Displaying the contents of a 3DO file (type.c)*

```

/*****
**
**  @(#) type.c 96/02/26 1.13
**
*****/

#include <:kernel:types.h>
#include <:kernel:operror.h>
#include <:file:fileio.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/*****

#define BUFFER_SIZE 1024

static void Type(const char *path, bool hex)
{
    RawFile *file;
    char      buffer[BUFFER_SIZE];
    int32     numBytes;
    uint32    numLines;
    Err       err;
    uint32    i;
    uint32    index;
    uint32    mod;
    char      hexLine[100];
    char      temp[10];
    uint32    start;

    /* Open the file as a byte-stream.  A buffer size of zero is
     * specified, which permits the file folio to choose an appropriate
     * amount of buffer space based on the file's actual block size.
     */

```



```
err = OpenRawFile(&file, path, FILEOPEN_READ);
if (err < 0)
{
    printf("File '%s' could not be opened: ", path);
    PrintfSysErr(err);
    return;
}

hexLine[0] = 0;
numLines    = 0;
index       = 0;
start       = 0;
while (TRUE)
{
    numBytes = ReadRawFile(file, buffer, BUFFER_SIZE);
    if (numBytes < 0)
    {
        printf("Error reading '%s': ", path);
        PrintfSysErr(numBytes);
        break;
    }
    else if (numBytes == 0)
    {
        break;
    }

    if (hex)
    {
        for (i = 0; i < numBytes; i++)
        {
            mod = index % 16;
            if (mod == 0)
            {
                if (index)
                    printf("%s", hexLine);

                sprintf(hexLine, "%04x: ", index);
                start = strlen(hexLine);
            }

            strcat(hexLine, "                                \n");

            sprintf(temp, "%02x", buffer[i]);
            hexLine[start + mod*2 + (mod / 4)] = temp[0];
            hexLine[start + mod*2 + (mod / 4) + 1] = temp[1];
        }
    }
}
```

```

        if (isgraph(buffer[i]))
            hexLine[start + mod + 39] = buffer[i];
        else
            hexLine[start + mod + 39] = '.';

        index++;
    }
}
else
{
    for (i = 0; i < numBytes; i++)
    {
        if (buffer[i] == '\r')
            buffer[i] = '\n';

        if (buffer[i] == '\n')
            numLines++;
    }

    printf("%*.*s", numBytes, numBytes, buffer);
}

if (hex)
{
    if (numBytes >= 0)
        printf("%s", hexLine);
}
else
{
    if (numBytes >= 0)
        printf("\n%u lines processed\n\n", numLines);
}

/* close the file */
CloseRawFile(file);
}

```

```

/*****

```

```

int main(int32 argc, char **argv)
{
    int32 i;
    bool  hex;

```

```
if (argc < 2)
{
    printf("Usage: type -hex <file1> [file2] [file3] [...]\n");
}
else
{
    hex = FALSE;
    for (i = 1; i < argc; i++)
    {
        if (strcasecmp(argv[i], "-hex") == 0)
        {
            hex = TRUE;
        }
        else if (argv[i][0] == '-')
        {
            printf("Ignoring unknown option '%s'\n",argv[i]);
        }
    }

    for (i = 1; i < argc; i++)
    {
        if (argv[i][0] != '-')
            Type(argv[i],hex);
    }
}

return 0;
}
```

Scanning the File System

Example 11-2 is a program listing of an application that scans the file system.

Example 11-2 Scanning the file system (Walker.c).

```

/*****
**
**  @(#) walker.c 96/02/26 1.12
**
*****/

#include <:kernel:types.h>
#include <:kernel:io.h>
#include <:kernel:operror.h>
#include <:file:filesystem.h>
#include <:file:directory.h>
#include <:file:directoryfunctions.h>
#include <:file:filefunctions.h>
#include <stdio.h>

/*****/

static void TraverseDirectory(Item dirItem)
{
    Directory      *dir;
    DirectoryEntry  de;
    Item           subDirItem;
    int32          entry;

    dir = OpenDirectoryItem(dirItem);
    if (dir)
    {
        entry = 1;
        while (ReadDirectory(dir, &de) >= 0)
        {
            printf("Entry #d:\n", entry);
            printf("  file '%s', type 0x%lx, ID 0x%lx",
                  de.de_FileName, de.de_Type,
                  de.de_UniqueIdentifier);
            printf(", flags 0x%lx\n", de.de_Flags);
            printf("  %d bytes, %d block(s) of %d byte(s) each\n",

```

```
        de.de_ByteCount,
        de.de_BlockCount, de.de_BlockSize);
printf("  %d avatar(s)", de.de_AvatarCount);
printf("\n\n");

if (de.de_Flags & FILE_IS_DIRECTORY)
{
    subDirItem = OpenFileInDir(dirItem, de.de_FileName);
    if (subDirItem >= 0)
    {
        printf("***** RECURSE *****\n\n");
        TraverseDirectory(subDirItem);
        printf("***** END RECURSION *****\n\n");
    }
    else
    {
        printf("**** RECURSION FAILED ***\n\n");
    }
    entry++;
}
CloseDirectory(dir);
}
else
{
    printf("OpenDirectoryItem() failed\n");
    CloseFile(dirItem);
}
}

/*****/

static Err Walk(const char *path)
{
    Item startItem;

    startItem = OpenFile((char *)path);
    if (startItem >= 0)
    {
        printf("\nRecursive directory scan from %s\n\n", path);
        TraverseDirectory(startItem);
        printf("End of %s has been reached\n\n", path);
        return 0;
    }
    else
```

```
    {
        printf("OpenFile(\"%s\") failed: ",path);
        PrintfSysErr(startItem);
        return startItem;
    }
}
```

```
/******
```

```
int main(int32 argc, char **argv)
{
    int32 i;

    if (argc <= 1)
    {
        /* if no directory name was given, scan the current directory */
        Walk(".");
    }
    else
    {
        for (i = 1; i < argc; i++)
            Walk(argv[i]);
    }

    return 0;
}
```

Listing the Contents of a Directory

Example 11-3 is a code listing that list the contents of a directory.

Example 11-3 Listing a directory (ls.c)

```

/*****
**
**  @(#) ls.c 96/02/26 1.17
**
*****/

#include <:kernel:types.h>
#include <:kernel:io.h>
#include <:kernel:operror.h>
#include <:file:filesystem.h>
#include <:file:filefunctions.h>
#include <:file:directory.h>
#include <:file:directoryfunctions.h>
#include <stdio.h>
#include <string.h>

/*****/

static void ListDirectory(const char *path)
{
    Directory      *dir;
    DirectoryEntry  de;
    Item            ioReqItem;
    int32           entry;
    int32           err, i;
    char            verRev[32];
    char            fullPath[FILESYSTEM_MAX_PATH_LEN];
    char            temp1[512], temp2[512];
    char            mtemp[5];
    IOInfo          ioInfo;
    Item            dirItem;

    /* open the directory for access */
    dirItem = OpenFile((char *)path);
    if (dirItem >= 0)
    {
        /* create an IOReq for the directory */
        ioReqItem = CreateIOReq(NULL, 0, dirItem, 0);
    }
}

```

```

if (ioReqItem >= 0)
{
    /* Ask the directory its full name. This will expand any aliases
     * given on the command-line into fully qualified pathnames.
     */
    memset(&ioInfo, 0, sizeof(ioInfo));
    ioInfo.ioi_Command      = FILECMD_GETPATH;
    ioInfo.ioi_Recv.iob_Buffer = fullPath;
    ioInfo.ioi_Recv.iob_Len   = sizeof(fullPath);
    err = DoIO(ioReqItem, &ioInfo);
    if (err >= 0)
    {
        /* now open the directory for scanning */
        dir = OpenDirectoryPath((char *)path);
        if (dir)
        {
            printf("\nContents of directory %s:\n\n", fullPath);
            entry = 1;
            while (ReadDirectory(dir, &de) >= 0)
            {
                memset(mtemp, 0, sizeof mtemp);
                memcpy(mtemp, &de.de_Type, 4);
                sprintf(temp2, "%5s", mtemp);
                sprintf(temp1, " %10d", de.de_ByteCount);
                strcat(temp2, temp1);
                sprintf(temp1, " %4d*%4d", de.de_BlockCount,
de.de_BlockSize);
                strcat(temp2, temp1);
                sprintf(temp1, " %3d", de.de_AvatarCount);
                strcat(temp2, temp1);
                if (de.de_Flags & FILE_HAS_VALID_VERSION) {
                    sprintf(verRev, "%3d.%d", de.de_Version,
                        de.de_Revision);
                    while (1) {
                        i = strlen(verRev);
                        if (i >= 8) {
                            break;
                        }
                        strcat(verRev, " ");
                    }
                    strcpy(temp1, verRev);
                } else {
                    strcpy(temp1, " ");
                }
                strcat(temp2, temp1);
                sprintf(temp1, " %8x", de.de_Flags);
                strcat(temp2, temp1);
            }
        }
    }
}

```



```
        sprintf(temp1, " %s\n", de.de_FileName);
    strcat(temp2, temp1);
    printf("%s", temp2);
        entry++;
    }
    CloseDirectory(dir);

    printf("\nEnd of directory\n\n");
}
else
{
    printf("OpenDirectory(\"%s\") failed\n",fullPath);
}
}
else
{
    printf("Unable to get full path name for '%s': ",path);
    PrintfSysErr(err);
}
DeleteIOReq(ioReqItem);
}
else
{
    printf("CreateIOReq() failed: ");
    PrintfSysErr(ioReqItem);
}
CloseFile(dirItem);
}
else
{
    printf("OpenFile(\"%s\") failed: ",path);
    PrintfSysErr(dirItem);
}
}

/*****

int main(int32 argc, char **argv)
{
    int32 i;

    if (argc <= 1)
    {
        /* if no directory name was given, scan the current directory */
        ListDirectory(".");
    }
    else
```

```
{
/* go through all the arguments */
for (i = 1; i < argc; i++)
    ListDirectory(argv[i]);
}

return 0;
}
```

The FSUtils Folio - File System Utilities

The FSUtils Folio provides a set of functions to do some high-level file system operations, such as copying entire directory trees, deleting entire directory trees, and performing some convenience operations involving path names.

Specifically, the FSUtils folio provides utilities to

- ◆ Copy a file or a whole directory tree from one directory to another on the same or different devices.
- ◆ Delete a directory tree within a file system.
- ◆ Determine the full, absolute pathname of a currently open file.
- ◆ Parse out and retrieve the final component of a pathname.
- ◆ Append a relative pathname onto another pathname (relative or absolute) to produce a new, composite pathname.

Copying a Directory Tree

To copy a directory hierarchy (i.e., tree) from one directory to another, you use a set of three functions.

- ◆ Call the `CreateCopyObj()` function to create a "copy object," which is the engine that controls the copy operation.
- ◆ Call the `PerformCopy()` function to perform the copy operation.
- ◆ Call the `DeleteCopyObj()` function to delete the copy object.

The input and destination directories can be on the same device or on different devices. Normally, the copying utilities would be used to copy a directory residing on a microcard to another directory on the same card or to a directory on another microcard.

None of these functions provide any user interface for the end-user. Instead, they provide hooks to call callback functions at strategic points during the copy operation. You create the callback functions, which provide the desired user interface to the end-user.

The reason that you use a set of three functions rather than a single copy function is to reduce the number of times the end-user has to insert and remove microcards when copying between two cards on a machine that has only one slot. A typical scenario on a one-slot machine could happen as follows:

1. The end-user selects "Copy Directory Tree" from a menu, browses the file system on the input device, and selects the input directory to be copied. This is all done using a user interface that you, the developer have provided.
2. Your code calls the `CreateCopyObj()` function, which does the following:
 - ◆ Sets up the copy engine.
 - ◆ Copies as much as possible of the input directory tree into RAM. This depends on the amount of RAM you have allowed for copying. Typically the whole input directory tree is stored into RAM at this point.
3. After the `CreateCopyObj()` function completes, your code then prompts the end-user to remove the input device, insert the destination device, and select the destination directory into which the input directory tree will be copied.
4. The end-user removes the input device, inserts the destination device, and selects the destination directory.
5. Your code calls the `PerformCopy()` function, which copies the input directory tree from RAM into the destination directory.

Note that since the input directory tree was already copied into RAM when the copy object was created, the end-user does not have to remove the destination device and reinsert the input device after selecting the destination directory.

If any interaction with the user is required during the copy, the copy engine calls a callback function that you have written and identified to the engine by means of a tag argument. Note that the copy engine does not provide any user interface for the end-user. It only provides hooks for callback functions that you can use to provide a user interface.

6. If the entire input directory was successfully copied, your code notifies the user and calls the `DeleteCopyObj()` function, which gets rid of the copy object and ends the copy operation.

If only part of the input directory could be stored in RAM, the appropriate callback function is called, which prompts the end-user to reinsert the input

device. This cycle continues until the entire input directory tree has been copied to the destination device.

On a machine with two slots, the copy is done directly from microcard to microcard with no intermediate removals and insertions needed. In this case, reading from the input and writing to the destination can be done asynchronously -- i.e., in parallel -- which can reduce the duration of the copy operation by as much as half.

Creating the Copy Object

To create the copy object, call the `CreateCopyObj()` function.

```
Err CreateCopyObj(CopyObj **co, const char *sourceDir,
                  const char *objName, const TagArg *tags)
```

You provide the following as arguments:

- ◆ A pointer to a location that will contain a pointer to the copy object.
- ◆ The pathname of the parent directory (`sourceDir`) that contains the file or directory tree that you want to copy.
- ◆ The name of the object (`objName` -- i.e., the file or the directory tree) that you want to copy from the parent directory to the destination.
- ◆ A pointer to an array of tags. These tags contain
 - ◆ Control information, such as
 - ◆ A pointer to the area where your callback functions can store local data (`COPIER_TAG_USERDATA (void *)`).
 - ◆ The maximum amount of RAM to use during the copy operation (`COPIER_TAG_MEMORYTHRESHOLD(uint 32)`). The default is 1M.
 - ◆ A flag indicating whether reads and writes can overlap on a two-slot machine (`COPIER_TAG_ASYNCHRONOUS (bool)`). The default is no.
 - ◆ Pointers to callback functions, such as
 - ◆ The function to call when the copy engine needs the source device (`COPIER_TAG_NEEDSOURCEFUNC (funcname)`).
 - ◆ The function to call when the copy engine needs the destination device (`COPIER_TAG_NEEDDESTINATIONFUNC (funcname)`).
 - ◆ The function to call during the copy to update your user interface with progress information (`COPIER_TAG_PROGRESSFUNC (funcname)`).
 - ◆ Functions to call when various error conditions are encountered (e.g., `COPIER_TAG_DESTINATIONFULLFUNC (funcname)`).

If you return a non-zero value from a callback function, the copy process will be stopped.

Note that you do not specify the destination directory at this time. Later, you will pass the pathname of the destination directory to the `PerformCopy()` function.

Performing the Copy

To provide the pathname of the destination directory and perform the copy operation, call the `PerformCopy()` function.

```
Err PerformCopy(CopyObj *co, const char *destinationDir)
```

You provide as arguments a pointer to the copy object and the pathname (relative or absolute) of the destination directory into which the input directory tree will be copied.

As `PerformCopy()` executes, it calls any appropriate callback functions that you pointed to when creating the copy object.

`PerformCopy()` returns a negative error code if the operation fails. Note that a failed operation may have copied some data to the destination.

Deleting the Copy Object

To clean up after a copy operation, call the `DeleteCopyObj()` function.

```
Err DeleteCopyObj(CopyObj *co)
```

You provide a pointer to the copy object. `DeleteCopyObj()` frees memory, flushes I/O buffers, closes files, and releases any other resources allocated by the copy object. While the function executes, it calls any appropriate callback functions that you pointed to when creating the copy object.

Deleting a Directory Tree

You can use the `DeleteTree()` function to delete a directory and all its subdirectories and files from any writable file system. Normally, you will only do this for file systems on microslot devices.

```
Err DeleteTree(const char *path, const TagArg *tags)
```

You provide as arguments the pathname of the directory to be deleted and a pointer to an array of tags.

The tags contain

- ◆ A pointer to the area where your callback functions can store local data (`DELETER_TAG_USERDATA (void *)`).
- ◆ Pointers to callback functions, such as
 - ◆ The function to call when the copier needs the device (`DELETER_TAG_NEEDMEDIAFUNC (funcname)`).
 - ◆ The function to call to update your user interface with progress information (`DELETER_TAG_PROGRESSFUNC (funcname)`).

- ◆ Functions to call when various error conditions are encountered (e.g., `DELETER_TAG_MEDIAPROTECTEDFUNC (funcname)`).

The deletion engine does not provide any user interface for the end-user. You use callback functions to provide any desired interaction with the user. If you return a non-zero value from a callback function, the copy process will be stopped.

Note that the deletion algorithm is implemented without recursion so you do not have to worry about running out of stack space on deep directories.

Determining the Full Pathname of an Open File

To determine the full, absolute path to a file that you have already opened, call the `GetPath()` function.

```
Err GetPath(Item file, char *path, uint32 numBytes)
```

You provide as arguments the file item for the opened file, a pointer to a buffer where the function will put the absolute pathname, and the size in bytes of the buffer.

The function puts the absolute pathname into the buffer and returns the length of the pathname or, if the operation fails, a negative error code.

You might use this function for a file that you originally opened using a relative pathname or for a file whose pathname you did not save when you originally opened it.

Extracting the Final Component of a Pathname

The `FindFinalComponent()` function provides a simple alternative to parsing when you need to extract the last component of a pathname – for example, if you want to retrieve the component “name” in the pathname “multi/component/name”.

```
char *FindFinalComponent(const char *path)
```

You provide as arguments the pathname (absolute or relative).

The function returns a pointer to the pathname’s last element. If the pathname only has a single element, the returned pointer will be to the start of the pathname string. For example, if the pathname is “/name”, the pointer will point to “/”.

Appending a Pathname to Another Pathname

The `AppendPath()` function is a simple convenience function that lets you concatenate a relative pathname onto the end of an existing pathname (relative or absolute). For example, you might want to add the path “appended/path” to the path “/base/path”, to produce a result of “/base/path/appended/path”.

```
Err AppendPath(char *path, const char *append,  
               uint32 numBytes)
```

You provide as arguments the base pathname, the pathname to append to the base, and the size in bytes of the buffer that will contain the result. The result will be placed in the buffer that currently contains the base.

`AppendPath()` handles slashes and relative and absolute pathnames intelligently.

Example 11-4 *Appending pathnames*

```
/first/second + third/fourth = /first/second/third/fourth  
  
/first/second/ + third/fourth = /first/second/third/fourth  
  
/first/second + /third/fourth = /third/fourth (see note below)
```

Note that an absolute pathname cannot be appended onto another pathname. In this case, the function just replaces the base with the absolute pathname that you attempted to append.

The Batt Folio and Date Folio

This chapter describes two folios that support the battery-backed clock, which exists in the M2 machine in addition to the microsecond and vertical blank timers.

This chapter contains the following topics:

Topic	Page Number
Introduction - The Battery-Backed Clock	171
The Batt Folio	172
The Date Folio	173

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Introduction - The Battery-Backed Clock

The M2 machine contains a battery-backed clock that provides a Gregorian (i.e., calendar) date in terms of year, month, day, hour, minute, and second. This is a separate clock from the microsecond and vertical blank timers. Unlike the timers, the battery-backed clock continues to operate and maintain time continuously after the machine is turned off. It is not reset to zero when the machine is rebooted.

Two folios support the battery-backed clock:

- ◆ The Batt folio allows you to read the current value of the clock and to set a new value.

- ◆ The Date folio allows you to convert back and forth between the Gregorian representation of date and time and the second-microsecond representation used by the microsecond timer. The Date folio also provides a function to validate an input Gregorian time to insure that it is within the limits that are valid for the battery-backed clock.

The clock is not initially set with a meaningful time, and there is not currently an external user interface that allows the end-user (e.g., game player) to read or set the clock's value. You, the developer, can use the functions described here to provide an appropriate interface to allow your end-users to read and set their local time in the clock.

The Batt Folio

Reading the Current Gregorian Date and Time

To read the current value of the battery-backed clock, use the `ReadBattClock()` function.

```
Err ReadBattClock(GregorianCalendar *gd)
```

As an argument, you pass a pointer to a structure that the function fills in with the current value expressed as a Gregorian date and time. This structure has the following format:

Example 12-1 *GregorianCalendar structure.*

```
typedef struct GregorianCalendar
{
    uint32 gd_Year;           /* 1..0xffff */
    uint16 gd_Month;          /* 1..12 */
    uint8  gd_Day; /* 1..28/29/30/31 (depends on month) */
    uint8  gd_Hour;           /* 0..23 */
    uint8  gd_Minute;         /* 0..59 */
    uint8  gd_Second;         /* 0..59 */
} GregorianCalendar;
```

The year is a binary value that will always be equal to or greater than 1993.

Setting the Current Gregorian Date and Time

To set the value of the battery-powered clock, use the `WriteBattClock()` function.

```
Err WriteBattClock(const GregorianCalendar *gd)
```

This function takes as input a pointer to the `GregorianCalendar` structure (shown above) and sets the clock to that value.

Note that the function will fail if the Gregorian value supplied is not within the valid range. To be within the valid range, the value of each field (year, month, day, hour, minute, second) must be valid. The year value must be equal to or greater than 1993. The day value must be appropriate for the month. For example, it cannot be 30 if the month is 2 (i.e., February).

The Date Folio

Converting from Gregorian to TimeVal Representation

To convert the Gregorian value provided by the battery-back clock to a `TimeVal` (i.e., seconds and microseconds) value use the `ConvertGregorianCalendarToTimeVal()` function. The `TimeVal` value is the same type of representation used by the microsecond timer.

```
Err ConvertGregorianCalendarToTimeVal(const GregorianCalendar *gd,
                                     TimeVal *tv)
```

This function takes a pointer to a `GregorianCalendar` structure (see Example 12-1 on page 172), which contains the value to convert, and a pointer to a `TimeVal` structure (shown below), which the function fills in.

Example 12-2 *TimeVal* structure.

```
typedef struct timeval
{
    int32 tv_Seconds;           /* seconds */
    int32 tv_Microseconds;      /* and microseconds */
} TimeVal;
```

The `TimeVal` value returned is in relation to a zero point of 0 hours:0 minutes:0 seconds on January 1, 1993. The `TimeVal` structure contains the number of seconds and microseconds from that zero point to the Gregorian date that you specified.

This function fails with an error code if the Gregorian date being converted is before January 1, 1993 or is in some other way invalid.

Converting from TimeVal to Gregorian Representation

To convert from a `TimeVal` representation to a Gregorian date and time, use the `ConvertTimeValToGregorianCalendar()` function.

```
Err ConvertTimeValToGregorian(const TimeVal *tv,  
    GregorianCalendar *gd)
```

The TimeVal structure contains the value in seconds and microseconds that you want to convert, and the GregorianCalendar structure is filled in by the function.

The GregorianCalendar date and time is calculated based on a TimeVal value of zero being equal to 0 hours:0 minutes:0 seconds on January 1, 1993. A TimeVal value of 60 seconds and 0 microseconds would convert to a GregorianCalendar date and time of 1 minute past midnight on January 1, 1993. This would be the GregorianCalendar date and time that you would get if you converted the TimeVal value of the microsecond timer one minute after rebooting the machine.

Validating a GregorianCalendar Date-Time Value

You can verify that a GregorianCalendar structure contains a permissible value by calling the ValidateDate() function.

```
Err ValidateDate(const GregorianCalendar *gd)
```

This function verifies that the date and time are on or after January 1, 1993 and that each individual field contains a valid value. The return code is positive or zero if the date-time is valid and negative if the date-time is invalid.

The Event Broker

This chapter shows how a task uses the event broker to work with interactive devices.

This chapter contains the following topics:

Topic	Page Number
About the Event Broker	176
Sending Messages Between Tasks and the Event Broker	178
The Process of Event Monitoring	182
Connecting a Task to the Event Broker	183
Monitoring Events Through the Event Broker	190
High-Performance Event Broker Use	199
Reconfiguring or Disconnecting a Task	206
Other Event Broker Activities	207
Event Broker Convenience Calls	214

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About the Event Broker

Working with an interactive device such as a control pad poses a challenge to traditional I/O architectures; a user can change the device's state at any time, so a task working with a device must monitor it constantly. Successful device monitoring requires a task to run a loop that continually polls the interactive device, or requires the interactive device driver to accept an I/O request but does not respond until it detects a change in the device. In the first case, the polling task uses a lot of processor time. In the second case, a task has to have several I/O requests pending to avoid losing any device changes while the driver returns an IOREq. Portfolio provides the *event broker*, a solution to interactive device I/O that requires neither polling nor multiple I/O requests.

The event broker is a task that constantly monitors activity on attached interactive devices. Currently, interactive devices are attached only to the control port on the 3DO system. The event broker monitors the control port to see what the devices are doing.

Activities on interactive devices are called *events*. Events include control pad presses and releases, key strokes, mouse rolling, plugging a new control device into the control port, unplugging a device from the control port, and many others. Events can occur on any hardware device attached to the control port. A hardware device is called a *pod*. The event broker always identifies the pod in which an event occurs.

The event broker starts when the system starts up, and stands between user tasks and the interactive pods on the control port. To work with interactive devices, a user task connects to the event broker and registers itself as a *listener*. A listener is a task that connects to the event broker to receive event notification and to pass data to pods. To become a listener, a task creates a message port for communication with the event broker. Then the task communicates its requests to the event broker with messages; the event broker responds to these messages with its own messages. I/O operations through the event broker are carried out through messages instead of IOREq structures.

Specifying and Monitoring Events

The first message a listener task sends to the event broker contains two *event masks*. A listener task uses the first event mask to specify which of the 256 different event types it is interested in hearing about. This mask is the *trigger mask*; it lists all event types that can trigger an event message to the listener. The second event mask is the *capture mask*; it lists all event types for which the event broker must check status whenever it sends an event message.

In each video field the event broker checks pods for events, a minimum of 60 times per second on NTSC systems, and 50 times per second on PAL systems. If any of the events it senses match an event in a listener task trigger mask, the event

broker composes an event message to send to the listener. That event message contains a pointer to an accompanying data block that contains information about all event types that are specified in either the trigger mask or the capture mask.

The event broker takes a "snapshot" of the pods' states whenever a triggering event occurs. For example, a listener can specify "control pad button pressed" as a trigger event and "mouse update" as a capture event.

When a control pad button is pressed, the event broker is triggered to send an event message to the listener. The event message's data block contains an event frame for the control pad buttons because pressing one of those buttons triggered the event message. This event frame contains a status bit for each control pad button, which indicates whether the button was pressed during that particular field.

In addition, there is also an event frame for the mouse because "mouse update" was specified as a capture event. In the mouse event frame, because it is for an "update" event, the status bits for the mouse indicate the current state of each button (i.e., up or down) and the current horizontal (x-axis) and vertical (y-axis) position regardless of whether the mouse was used during that field. Note that specifying a "pressed" event for the trigger event and an "update" event for the capture event is common practice.

Because the event broker notifies listeners of events via messages, a listener task can enter wait state for an event broker message, just as a task waits for any other message. This allows synchronous I/O with interactive pods. A listener can also continue execution after sending an event broker message, which allows asynchronous I/O. In either case, whenever a listener task receives a message from the event broker, it must reply to the event broker and say the message is processed and can be reused. If the listener task does not reply, the event message queue can fill up, and it can lose events in which it was interested.

Working With Input Focus

Input focus is an important concept to both 3DO users and 3DO tasks. When several tasks occupy the display at the same time, input focus determines which task should respond to the user's actions. For example, a pinball game occupies one part of the display, while a CD-ROM encyclopedia occupies another part. When the user presses the "left" direction of the control pad cross, does the left pinball flipper flip, or does the reader for the encyclopedia move back a page? It depends on which program has the input focus. If the user chose the reader earlier (via menu selection, for example) so that the reader is now the active task, the reader holds the input focus and responds to the control pad.

The event broker sends events to the appropriate focus holder—if connected tasks are set up to pay attention to focus. Whenever a task connects to the event broker, it can request to be one of three listener types:

- ◆ A **focus-dependent listener**, which only receives its specified events from the event broker when the event broker knows that the task has the input focus.
- ◆ A **focus-independent listener**, which receives all of its specified events from the event broker at all times.
- ◆ A **focus-interested listener**, which receives all of its specified events when it has the focus, and only non-user interface type events when it does not have the focus.

Tasks such as the encyclopedia reader and the pinball game mentioned above are usually focus-dependent listeners because they only respond when the user selects the task. Tasks that offer a program-selection menu to the user are usually focus-independent because they must respond at any time to user requests, even if they do not hold the focus.

Reconfiguring or Disconnecting an Event Broker Connection

As a listener task works with the event broker, it can send a message requesting a new configuration at any time. The new configuration can reset the task's focus dependence, specify new capture and trigger events, set a new message port, or change other connection parameters. A listener task can also request to be disconnected from the event broker so that it will no longer be notified of interactive events.

Other Event Broker Activities

Although the event broker's main activity is monitoring and reporting on interactive device events, it can also:

- ◆ Provide lists of connected devices and listener tasks
- ◆ Send and receive I/O data from interactive devices
- ◆ Send generic commands to interactive devices

Sending Messages Between Tasks and the Event Broker

All communication between the event broker and connected listener tasks is through messages (a process discussed in earlier kernel chapters). These messages, which pass from listener to event broker or from event broker to listener, are called *event broker messages*. Before programming with the event broker, you should understand how these messages work.

Message Flavors

Each event broker message comes in a *flavor* that identifies the purpose of the message. The flavor of the message determines the type (or types) of data structures contained in the data block, and specifies how the message recipient

should handle the message. The data structure `EventBrokerHeader` is always the first field of the first data structure within a message data block. Its definition is shown below:

```
typedef struct EventBrokerHeader
{
    enum EventBrokerFlavor ebh_Flavor;
} EventBrokerHeader;
```

`EventBrokerHeader` indicates the flavor of the message. Table 13-1 shows the available flavors.

Table 13-1 *Event broker message flavors.*

Event Broker Flavor	Operation Requested
EB_NoOp	No operation requested.
EB_Configure	(Task to EB) Connect this task to the event broker and register its configuration.
EB_ConfigureReply	Event broker reply to EB_Configure.
EB_EventRecord	(EB to task) The events listed in the data block occurred in this field and are events in which the task is interested.
EB_EventReply	A listener's reply to EB_EventRecord.
EB_SendEvent	(Task to EB) An event has happened within a task. This command is not currently implemented.
EB_SendEventReply	Event broker reply to EB_SendEvent.
EB_Command	This operation is not currently determined.
EB_CommandReply	Event broker reply to EB_Command.
EB_RegisterEvent	(Task to EB) Register this custom event name and assign an event number to that name. This command is not currently implemented.
EB_RegisterEventReply	Event broker reply to EB_RegisterEvent.
EB_GetListeners	(Task to EB) Tell the requesting task which tasks are connected to the EB.

Table 13-1 *Event broker message flavors. (Continued)*

Event Broker Flavor	Operation Requested
EB_GetListenersReply	Event broker reply to EB_GetListeners.
EB_SetFocus	(Task to EB) Assign the input focus to a specified task.
EB_SetFocusReply	Event broker reply to EB_SetFocus.
EB_GetFocus	(Task to EB) Tell the requesting task which task has the current input focus.
EB_GetFocusReply	Event broker reply to EB_GetFocus.
EB_ReadPodData	(Task to EB) Send requesting task data from the specified pod. This command is not currently implemented.
EB_ReadPodDataReply	Event broker reply to EB_ReadPodData.
EB_WritePodData	(Task to EB) Write the enclosed data to the specified pod. This command is not currently implemented.
EB_WritePodDataReply	Event broker reply to EB_WritePodData.
EB_LockPod	(Task to EB) Lock the specified pod so only the requesting task can issue commands or write data to it. This command is not currently implemented.
EB_LockPodReply	Event broker reply to EB_LockPod.
EB_UnlockPod	(Task to EB) Unlock the specified pod so that other tasks can issue commands or write data to it. This command is not currently implemented.
EB_UnlockPodReply	Event broker reply to EB_UnlockPod.
EB_IssuePodCmd	(Task to EB) Issue this generic command to the specified pod.
EB_IssuePodCmdReply	Event broker reply to EB_IssuePodCmd.
EB_DescribePods	(Task to EB) Describe pods attached to the control port.
EB_DescribePodsReply	Event broker reply to EB_DescribePods.

Table 13-1 Event broker message flavors. (Continued)

Event Broker Flavor	Operation Requested
EB_MakeTable	(Task to EB) Create a pod table and return a pointer to it.
EB_MakeTableReply	Event broker reply to EB_MakeTable

With the exception of EB_NoOp, event broker messages come in pairs; every event broker operation request is answered with a reply from either the event broker or the receiving task. The reply confirms an operation's execution (or failure to execute). A reply message can carry information requested in the operation request.

All operation requests are sent from a listener task to the event broker with one exception: EB_EventRecord, which the event broker sends to a listener to tell it what specified events happened during a field.

Message Types

Portfolio supports three types of messages:

- ◆ Standard
- ◆ Small
- ◆ Buffered

To communicate with the event broker, a listener task can use either standard or buffered messages; small messages do not work.

Whenever a listener task asks the event broker for information, it must use a buffered message. For example, if the EB_DescribePods command is sent, the event broker puts the pod information in the buffer associated with the message and returns the message to the listener task.

If the listener task is giving information to the event broker, for example using an EB_WritePodData command, the task can use either a standard (pass by reference) or buffered (pass by value) message. The event broker reads the data from the data block, but does not modify it.

When a task replies to event broker messages, such as returning an EB_EventRecord message, it should use the ReplyMsg() function. Always provide NULL for the data pointer, and 0 for the data size arguments to ReplyMsg().

Flavor-Specific Message Requirements

Each message flavor requires a specific data structure (or set of data structures) to accompany it. The data structures are defined in the include file `<misc/event.h>` (see “Reading Event Data” on page 194). The data structures are also discussed in the sections of this chapter that describe different event broker operations.

Operation requests and replies are sent with the same message. The event broker or receiving task fills in the appropriate reply data, and returns the message.

When requesting data from the event broker, a listener task must use a buffered message to hold the response. Because the reply data from an operation can be extensive, the message should have a large enough data block for the event broker to fill in data. If the data block is not large enough, the event broker puts no data in the buffer and returns an error.

Note that a reply message cannot copy useful data to the data blocks if the reply does not require large amounts of data. For example, a listener task can use an `EB_GetFocus` message to ask the event broker to report which task has the input focus (a process described in “Working With Input Focus” on page 210). The event broker only needs to store the item number of the task with focus in the error field of the `EB_GetFocus` message and return the message without copying any data to the data block. In such a case, a task can use a standard message to request information instead of a buffered message because the buffer is not used.

The Process of Event Monitoring

Although event broker messages come in enough flavors to allow a task to carry out a variety of I/O operations through the event broker, the event broker’s main function is to monitor events in user-interface devices and report the events to listening tasks. From a listener task’s point of view, the process of event monitoring has the following steps:

1. Connecting to the event broker
 - ◆ Create a message port
 - ◆ Create a configuration message
 - ◆ Create a configuration block
 - ◆ Send the configuration message to the event broker
 - ◆ Receive the event broker’s connection reply message
2. Monitoring events through the event broker
 - ◆ Wait for event messages on the allocated message port
 - ◆ Read an event message data block
 - ◆ Process event data block information (if desired)
 - ◆ Reply to the event broker for event message receipt
3. Changing configuration (if desired) or disconnecting from the event broker

- ◆ Send a new configuration data structure to the event broker

Each of these steps use event broker data structures and message flavors.

Connecting a Task to the Event Broker

Before a task can work with the event broker, it must connect to the event broker by sending a message that requests connection as a listener. To do so, the task sends a message of the flavor `EB_Configure`, using Portfolio's standard message-sending procedure. The message also specifies input focus, and indicates the event types in which the task is interested.

Creating a Message Port

Before creating a configuration message, a listener task must create its own message port to use as a reply port where the event broker will send messages. To do so, the listener uses the `CreateMsgPort()` call, which returns the item number of the new message port.

Creating a Configuration Message

After a listener task creates a message port, it must create a message using `CreateMsg()` or `CreateBufferedMsg()`. Both calls accept a string that names the message, supplies an optional priority number for the message, and provides the item number of the listener message port. `CreateBufferedMsg()` also takes the size of the buffer to allocate with the message. Both calls return the item number of the newly created message.

Creating a Configuration Block

To turn the newly created message into a configuration request, the listener task must create a configuration data block, which consists of the `ConfigurationRequest` structure, shown in Example 13-1.

Example 13-1 *ConfigurationRequest structure.*

```
typedef struct ConfigurationRequest
{
    EventBrokerHeader cr_Header;          /* { EB_Configure } */
    enum ListenerCategory cr_Category; /* focus, monitor, or
                                         /* hybrid */

    uint32 cr_TriggerMask[8];             /* events to trigger on */
    uint32 cr_CaptureMask[8];            /* events to capture */
    int32 cr_QueueMax;                   /* max # events in transit */
    uint32 rfu[8];                       /* must be 0 for now */
} ConfigurationRequest;
```

The listener task must fill in the following fields to set the parameters of its configuration request:

- ◆ `cr_Header` is an `EventBrokerHeader` data structure in which the event flavor `EB_Configure` must be set.
- ◆ `cr_Category` defines the listener type of input focus. It has four possible values:
 - ◆ `LC_FocusListener` sets the task to be a focus-dependent listener. It must have the focus to receive any events.
 - ◆ `LC_FocusUI` sets the task to be a focus-interested listener. It must have the focus to receive user interface events, but gets all other event types regardless of focus.
 - ◆ `LC_Observer` sets the task to be a focus-independent listener. It gets all event types regardless of focus.
 - ◆ `LC_NoSeeUm` sets the task to ignore all events completely, the equivalent of not connecting to the event broker.
- ◆ `cr_TriggerMask` and `cr_CaptureMask` contain the event masks that specify events in which the task is interested. The trigger mask specifies each event type that will trigger the event broker to notify the task; the capture mask specifies each event type that the event broker reports on whenever it notifies the task of an event triggering. These masks are discussed in “The Event Broker Trigger Mechanism” on page 190.
- ◆ `cr_QueueMax` specifies the maximum number of event reports that the event broker posts at any one moment in the task’s event queue. This value can be set to the constant `EVENT_QUEUE_MAX_PERMITTED`, which sets the currently defined maximum number of events in the queue (20 in this release), or the constant `EVENT_QUEUE_DEFAULT`, which sets the currently defined default number of events (three in this release). The queue must have a minimum depth of at least two events.

- ◆ The `rfu` field is reserved for future use, and currently must be set to 0.

Event Types

The 256 different event types available through the event broker are divided into two main bodies:

- ◆ **System events.** 128 system-defined generic events.
- ◆ **Custom events.** 128 task-defined events primarily used for communicating events in one event broker listener to other listeners. Custom events are not currently implemented.

Both system events and custom events are divided into two types that are defined by the way the event broker reports them to focus-interested tasks:

- ◆ **UI events**, which are not reported to a focus-interested task if a task does not hold the focus. The event broker provides 64 system and 64 custom UI events.
- ◆ **Non-UI events**, which are reported to a focus-interested task even if the task does not hold the focus. The event broker provides 64 system and 64 custom non-UI events.

Table 13-2 shows how the 256 different bit values in a bit mask are divided into system and custom events, and then UI and non-UI events. Note that some events are indicated as defined but not yet implemented. Note also that custom event types are not supported in this release of Portfolio.

Table 13-2 *Flag Bit constants defined in event.h.*

System UI Event Type Constants	Event Definition
<code>EVENTBIT0_ControlButtonPressed</code>	A control pad button was pressed.
<code>EVENTBIT0_ControlButtonReleased</code>	A control pad button was released.
<code>EVENTBIT0_ControlButtonUpdate</code>	A control pad button was pressed or released, or control button information may have been lost in an event queue overflow. This event lets you determine the current state of the control buttons.
<code>EVENTBIT0_ControlButtonArrived</code>	Data from a control pad button has arrived (it arrives every field).
<code>EVENTBIT0_MouseButtonPressed</code>	A mouse button was pressed.
<code>EVENTBIT0_MouseButtonReleased</code>	A mouse button was released.

Table 13-2 *Flag Bit constants defined in event.h. (Continued)*

System UI Event Type Constants	Event Definition
EVENTBIT0_MouseUpdate	A mouse button was pressed or released, the mouse has moved, or mouse info may have been lost in an event queue overflow. This event lets you determine the current state of the mouse buttons and the mouse's current position.
EVENTBIT0_MouseMoved	A mouse has moved.
EVENTBIT0_MouseDataArrived	Data from a mouse has arrived (it arrives every field).
EVENTBIT0_GunButtonPressed	A gun button was pressed. This is not currently implemented.
EVENTBIT0_GunButtonReleased	A gun button was released. This is not currently implemented.
EVENTBIT0_GunUpdate	A gun button was pressed or released, or gun button information may have been lost in an event queue overflow. This event lets you determine the current state of the gun buttons. This is not currently implemented.
EVENTBIT0_GunDataArrived	Data from a gun has arrived (it arrives every field). This is not currently implemented.
EVENTBIT0_KeyboardKeyPressed	A keyboard key was pressed. This is not currently implemented.
EVENTBIT0_KeyboardKeyReleased	A keyboard key was released. This is not currently implemented.
EVENTBIT0_KeyboardUpdate	A keyboard key was pressed or released, or keyboard information may have been lost in an event queue overflow. This event lets you determine the current state of the keyboard keys. This is not currently implemented.
EVENTBIT0_KeyboardDataArrived	Data from a keyboard has arrived (it arrives every field). This is not currently implemented.
EVENTBIT0_CharacterEntered	A character was entered. This is not currently implemented.

Table 13-2 *Flag Bit constants defined in event.h. (Continued)*

System UI Event Type Constants	Event Definition
EVENTBIT0_GivingFocus	This task was given focus.
EVENTBIT0_LosingFocus	This task lost focus.
EVENTBIT0_LightGunButtonPressed	A light gun button was pressed.
EVENTBIT0_LightGunButtonReleased	A light gun button was released.
EVENTBIT0_LightGunUpdate	A light gun button was pressed or released, or light gun button information may have been lost in an event queue overflow. This event lets you determine the current state of the light gun buttons
EVENTBIT0_LightGunFireTracking	Data from a light gun is being tracked.
EVENTBIT0_LightGunDataArrived	Data from a light gun has arrived (it arrives every field).
EVENTBIT0_StickButtonPressed	A joystick button was pressed.
EVENTBIT0_StickButtonReleased	A joystick button was released.
EVENTBIT0_StickUpdate	A joystick button was pressed or released, or joystick button information may have been lost in an event queue overflow. This event lets you determine the current state of the joystick buttons and the joystick's position.
EVENTBIT0_StickMoved	A joystick has moved.
EVENTBIT0_StickDataArrived	Data from a joystick has arrived (it arrives every field).
EVENTBIT0_IRKeyPressed	An IR key was pressed and a button code is returned. This is device specific. This is not currently implemented.
EVENTBIT0_IRKeyReleased	An IR key was released and a button code is returned. This is device specific. This is not currently implemented.

Table 13-3 shows the flag bit constants defined in *event.h* for the system events, along with the meaning for each of the event types. These event type constants define flags for system-defined events, each of which can occur during a single control port polling.

Table 13-3 *Event type constants that define flags for system-defined events.*

System Non-UI Event Type Constants	Event Definition
EVENTBIT2_DeviceChanged	A device was added or removed.
EVENTBIT2_FileSystemMounted	A file system was mounted.
EVENTBIT2_FileSystemOffline	A file system went off-line.
EVENTBIT2_FileSystemDismounted	A file system was dismounted.
EVENTBIT2_ControlPortChange	A new device was plugged into or disconnected from the control port.
EVENTBIT2_PleaseSaveAndExit	This task was requested to save current status and exit.
EVENTBIT2_PleaseExitImmediately	This task was requested to exit immediately.
EVENTBIT2_EventQueueOverflow	This task's event queue has overflowed and the task has lost events.
EVENTBIT2_DetailedEventTiming	Detailed event timing data (exact time of event from microsecond timer) available in DetailedEventData structure. You might want this on rare occasions for particularly high timing resolution.

Setting the Trigger and Capture Masks

Each of the masks in the ConfigurationRequest data structure is an 8-element array of 32-bit words that provides 256 bits, 1 bit for each event type. To set a mask, the task logically ORs the desired constants defined for each word of the array, and stores the results in the appropriate word. The constant name identifies the appropriate word for storage. For example, all the desired flag constants starting with EVENTBIT0 must be logically ORed and then stored in word 0 of the array. All the desired flag constants starting with EVENTBIT2 must be logically ORed and then stored in the second word of the array.

Sending the Configuration Message

Once the `ConfigurationRequest` data structure has been created and filled in, the listener task must send the message to the event broker. To do this, the task first must find the event broker message port using the `FindMsgPort()` call, as shown in Example 13-2.

Example 13-2 `FindMsgPort()`.

```
{
Item ebPortItem;

    ebPortItem = FindMsgPort(EventPortName);
    if (ebPortItem < 0)
    {
        /* big trouble, the event broker can't be found! */
    }
}
```

After the listener task has the event broker message port item number, the task sends its configuration message using `SendMsg()`:

```
SendMsg(ebPortItem, msg, &configuration, sizeof(configuration))
```

The `SendMsg()` call accepts four arguments: the item number of the event broker port, the item number of the message the task has created, a pointer to the `ConfigurationRequest` data structure the task has initialized, and the size of that data structure.

When the event broker receives the configuration message, it adds the task as a listener. The event broker reads the reply port contained in the message and uses that port whenever it needs to communicate with the listening task. The event broker also reads the `ConfigurationRequest` data structure and sets up the listener's configuration accordingly.

Receiving the Configuration Reply Message

When the event broker finishes processing the configuration message, it changes the message to a configuration reply message. That message contains a pointer to a single data structure: an `EventBrokerHeader` with the value `EB_ConfigureReply`. The event broker returns the message to the task requesting configuration. The receiving task checks the configuration message's result field (`msg_Result`, which is cast to an `Err` type) to see if the operation was successful or not. If the value is a negative number, it is an error code, and the configuration was not successful (the task was not connected). If the value is 0 or a positive number, the configuration was successful and the task is connected.

Monitoring Events Through the Event Broker

Waiting for Event Messages on the Reply Port

After a task is connected as a listener, it can choose either of the following methods for event notification:

- ◆ The task can enter wait state until it receives an event broker message on the reply port (synchronous I/O).
- ◆ The task can continue to execute, and pay attention to the event broker only when the task receives notification of a message received on the reply port (asynchronous I/O).

The Event Broker Trigger Mechanism

While listeners wait for event notification, the event broker checks the pods at least once per field for events. The event broker checks occurring events against the trigger mask of each connected listener. If an event matches a set bit in a listener's trigger mask and the listener has the input focus (if required), the event broker sends an event notification to the listener.

When the event broker sends an event notification, it uses its own event messages, which it creates as necessary. After the event broker creates an event message, it copies the information into the message's data block. The information contains a report of the status of each event type that matches a set bit in either the listener of the trigger or capture masks. You can think of the event message data block as a snapshot of the status of all listener-specified events during the field. Only events specified in the trigger mask can trigger the snapshot, but the status of all events specified in either mask is reported in the message.

For example, consider a listener that specifies the "control pad button pressed" event type in its trigger mask, and the "mouse update" event type in its capture mask. When any button on the control pad is pressed during a field, the event broker prepares an event message for the listener.

The event message reports whether each control pad button was pressed during the field and also reports the current status of the mouse buttons (up or down) and the current mouse position (horizontal and vertical). If no control pad buttons had been pressed, the event broker would not have sent an event message to the listener -- even if the mouse had been pressed or moved -- because the mouse event was in the capture mask but not in the trigger mask.

Retrieving and Replying to Event Messages

After a listener has been notified of an incoming event message from the event broker, it must retrieve the message to read the reported events. To do so, it uses the `GetMsg()` call. The listener can then read the message's data block (described in "Reading an Event Message Data Block" below), and either act on the data or store it for later use.

After a listener has processed the event data in an event message, the listener must reply to the event broker. Its reply tells the event broker that the event message is once again free for use. If the listener does not reply to an event message, the event broker assumes that the message was not processed. Each unprocessed event message is one element in the listener's event queue. When the number of unprocessed event messages exceeds the maximum event queue depth, the event broker stops reporting events to the listener. Consequently, the listener loses events until it can process and free up messages in its queue. If the listener asks to be notified of an event queue overflow, the event broker reports it as soon as an event message is free. The update event types (`EB_ControlButtonUpdate`, `EB_MouseUpdate`, and so on) are all reported as occurring, because an event queue overflow is considered an update.

To reply to an event message and return it to the event broker, a listener uses the `ReplyMsg()` call. Always pass `NULL` as the data pointer, and 0 as the data size parameter to `ReplyMsg()`.

Reading an Event Message Data Block

When a listener receives an event message from the event broker, a pointer to the event data block is in the `msg_DataPtr` field of the message structure, and the size in bytes of the data block in the `msg_DataSize` field. To interpret the event data stored in the event message, the listener task must be able to read the data structures used in the data block.

Event Data Block Structure

When the event broker reports events in an event data block, it can report one or more events. Therefore, it must have a flexible data arrangement within the data block to report any number of events. An event data block starts with an `EventBrokerHeader` structure that specifies the message flavor as `EB_EventRecord` followed with one or more `EventFrame` structures, each containing the report of a single event. (Each event report is called an *event frame*.) The final structure in the data block is a degenerate `EventFrame` data structure, which indicates the end of the event frames.

The EventFrame Data Structure

The `EventFrame` data structure is defined in `<misc/event.h>` as follows:

Example 13-3 EventFrame structure

```
typedef struct EventFrame
{
    uint32 ef_ByteCount;          /*total size of EventFrame */
    uint32 ef_SystemID;           /*3DO machine ID, or 0=local*/
    TimeValVBL ef_SystemTimeStamp; /*event-count timestamp*/
    int32 ef_Submitter;           /*Item of event sender, or 0*/
    uint8 ef_EventNumber;         /*event code, [0,255]*/
    uint8 ef_PodNumber;           /*CP pod number, or 0*/
    uint8 ef_PodPosition;
                                /* CP position on daisychain, or 0 */
    uint8 ef_GenericPosition;
                                /* Nth generic device of type, or 0 */
    uint8 ef_Trigger;             /*1 for trigger, 0 for capture*/
    uint8 rful[3];
    uint32 rfu2;
    uint32 ef_EventData[1];       /* first word of event data */
} EventFrame;
```

- ◆ ef_ByteCount gives the total size of the EventFrame in bytes. Because the event data at the end of the frame varies in size depending on the event, this value changes from frame to frame.
- ◆ ef_SystemID reports the ID number of the 3DO unit where this event occurred. This value is useful if two or more 3DO units are linked together (as they might be for networked games). If the event occurred on the local 3DO unit, this value is set to 0.
- ◆ ef_SystemTimeStamp is the exact system vblank count when the event broker recorded this event.
- ◆ ef_Submitter gives the item number of the event sender. This value is important when multiple listener tasks tied into the event broker send events to each other. This item number is the item number of the sending task. If the event comes from a pod and not a task, this field is set to 0.
- ◆ ef_EventNumber is an event code from 0 to 255 that identifies the generic type of event. These event types are the same as the event types identified within a configuration event mask. The include file *event.h* defines a constant for each type as shown in Table 13-4.
- ◆ ef_PodNumber gives the unique pod number of the pod where an event originated. This number is constant and is assigned when a pod is first plugged into the 3DO unit. It does not change if the pod changes its position in the control port daisy chain. This value is set to 0 if the event did not

originate in a pod (for example, an event generated by another task or the CD-ROM drive).

- ◆ `ef_PodPosition` gives the position of the event-reporting pod in the control port daisy chain. Numbering starts with 1 for the first pod, and continues consecutively the further the chain extends from the 3DO unit. If the event did not originate in a pod, this value is set to 0.
- ◆ `ef_GenericPosition` gives the daisy-chain position of the event-reporting pod among identical generic device types in the daisy chain. A generic device type is a functional description of the pod or part of the pod, and currently includes the following defined types:
 - ◆ `POD_IsControlPad`
 - ◆ `POD_IsMouse`
 - ◆ `POD_IsGun`
 - ◆ `POD_IsGlassesCtrlr`
 - ◆ `POD_IsAudioCtrlr`
 - ◆ `POD_IsKeyboard`
 - ◆ `POD_IsLightGun`
 - ◆ `POD_IsStick`
 - ◆ `POD_IsIRController`

For example, an event occurs in the second control pad in the daisy chain, in which case this value is 2, or in the fourth photo-optic gun in the daisy chain, in which case this value is 4. This value is set to 0 if the event did not occur in a generic device.

A single pod attached to the 3DO unit can have more than one generic device type. For example, a single pod can be a combination control pad and audio controller. In that case, when an event occurs on its control pad buttons, the generic position listed in the event frame is that of a control pad among other control pads connected to the 3DO unit.

- ◆ `ef_Trigger` identifies an event as one that triggered the event notification or as a captured event that did not trigger the notification. If the value is 1, the event was a triggering event; if the value is 0, the event is a captured event.
- ◆ `ef_EventData` is the first word of a data structure that contains data about an event. The rest of the data structure follows this field in memory. The definition of the data structure depends entirely on the generic device where the event occurred. The data structures are described in “Reading Event Data” on page 194.

The Final (Degenerate) Event Frame

The final event frame within an event data block must be a degenerate event frame, which contains the value 0 for the `ef_ByteCount` field. The rest of the `EventFrame` data structure is not present.

Reading Event Data

The event data structure at the end of each full event frame is a data structure that determines the type of device reporting the event. For example, an event occurring on a control pad uses a structure designed to report control pad data. An event occurring on a mouse uses a structure designed to report mouse data.

Control Pad Data Structure

The data structure that reports control pad data is `ControlPadEventData`, defined in *event.h* as follows:

Example 13-4 `ControlPadEventData` structure

```
typedef struct ControlPadEventData {
    uint32      cped_ButtonBits; /* left justified, zero fill */
    uint8       cped_AnalogValid; /* nonzero if remaining fields
                                   valid */
    uint8       cped_StickX;      /* value 0 to 255 */
    uint8       cped_StickY;      /* value 0 to 255 */
    uint8       cped_LeftShift;   /* value 0 to 15 */
    uint8       cped_RightShift;  /* value 0 to 15 */
    uint8       cped_Shuttle;     /* value 0 to 255 */
    uint8       rfu_mbz[2];       /* filler */
} ControlPadEventData;
```

- ◆ `cped_ButtonBits` contains a value whose bits tell the status of each control pad button during the field. The constants defining these flag bits are as follows:
 - ◆ `ControlDown` — the down arm of the control pad cross.
 - ◆ `ControlUp` — the up arm of the control pad cross.
 - ◆ `ControlRight` — the right arm of the control pad cross.
 - ◆ `ControlLeft` — the left arm of the control pad cross.
 - ◆ `ControlA` — the A button.
 - ◆ `ControlB` — the B button.
 - ◆ `ControlC` — the C button.
 - ◆ `ControlD` — the D button (on new M2 pad only).
 - ◆ `ControlE` — the E button (on new M2 pad only).
 - ◆ `ControlF` — the F button (on new M2 pad only).
 - ◆ `ControlStart` — the P (play/pause) button.
 - ◆ `ControlX` — the X (stop) button (on older, Opera pads only).
 - ◆ `ControlTrigger` — the Trigger button (on new M2 pad only).
 - ◆ `ControlLeftShift` — the left shift button.

- ◆ `ControlRightShift` — the right shift button.

The meaning of these bits varies for each type of control pad event:

- ◆ For the `EVENTNUM_ControlButtonPressed` event, 1 means the button was pressed since the last field, and 0 means the button was not pressed.
- ◆ For the `EVENTNUM_ControlButtonReleased` event, 1 means the button was released since the last field, and 0 means the button was not released.
- ◆ For the `EVENTNUM_ControlButtonUpdate` event, 1 means the button is down, and 0 means the button is up. See “A Note on the `EVENTNUM_ControlButtonUpdate` Event” on page 195.
- ◆ For the `EVENTNUM_ControlButtonArrived` event, 1 means the button is down, and 0 means the button is up. See “A Note on the `EVENTNUM_ControlButtonArrived` Event” on page 196.
- ◆ `cped_AnalogValid` indicates that the following analog fields have been set with valid information. Note that this bit will only be set for the new M2 pads.
- ◆ `cped_StickX` is the analog value for the joystick X axis. Note that the maximum may actually be less than 255, and you should not assume a particular maximum.
- ◆ `cped_StickY` is the analog value for the joystick Y axis. Note that the maximum may actually be less than 255, and you should not assume a particular maximum.
- ◆ `cped_LeftShift` is the analog value for the Left Shift button.
- ◆ `cped_RightShift` is the analog value for the Right Shift button.
- ◆ `cped_Shuttle` is the analog value for the Shuttle control ring.

A Note on the `EVENTNUM_ControlButtonUpdate` Event

The `EVENTNUM_ControlButtonUpdate` event message is sent to you under two kinds of circumstances:

1. You subscribe to the `EVENTNUM_ControlButtonUpdate` event, and a control pad button changes state -- i.e., is pressed or released.
Typically, you only want to know when the button is pressed and would not subscribe to the `EVENTNUM_ControlButtonUpdate` event because it would provide too much unneeded data to your process.
2. You subscribe to the `ControlButtonPressed` event, and a control pad button is pressed repeatedly so quickly that your process's event queue overflows and events are lost.

In this case, the event broker sends you a `EVENTNUM_ControlButtonUpdate` event message to tell you that you are out of synch and what the current status is of each of the control buttons -- i.e.,

whether each is currently up or down. A value of 1 means the button is down, and a value of 0 means the button is up.

A Note on the EVENTNUM_ControlButtonArrived Event

This event is triggered every time data from a control pad button arrives, which happens during every field. Typically, you would not subscribe to this event because it would provide too much unneeded data to your process.

Mouse and Trackball Data

The data structure used to report mouse and trackball data is `MouseEventData`, defined in *event.h* as follows:

```
typedef struct MouseEventData
{
    uint32    med_ButtonBits;    /* left justified, zero fill */
    int32     med_HorizPosition;
    int32     med_VertPosition;
} MouseEventData;
```

- ◆ `med_ButtonBits` contains a value whose bits tell which mouse button (or buttons) generated the event. The constants defining these flag bits are as follows:
 - ◆ `MouseLeft` - the left mouse button.
 - ◆ `MouseMiddle` - the middle mouse button.
 - ◆ `MouseRight` - the right mouse button.
 - ◆ `MouseShift` - the mouse's shift button.

A 3DO mouse currently has three buttons: left, middle, and right. The fourth constant provides for an extra shift button if one ever appears on a 3DO mouse.

The meaning of the above bits varies for each type of mouse event:

- ◆ For the `EVENTNUM_MouseButtonPressed` event, 1 means the button was pressed since the last field, and 0 means the button was not pressed.
- ◆ For the `EVENTNUM_MouseButtonReleased` event; 1 means the button was released since the last field, and 0 means the button was not released.
- ◆ For the `EVENTNUM_MouseUpdate` event; 1 means the button is down, and 0 means the button is up. This event works in a similar way to the `EVENTNUM_ControlButtonUpdate` event. See "A Note on the `EVENTNUM_ControlButtonUpdate` Event" on page 195.
- ◆ For the `EVENTNUM_MouseDataArrived` event; 1 means the button is down, and 0 means the button is up. This event works in a similar way to

the `EVENTNUM_ControlButtonArrived` event. See “A Note on the `EVENTNUM_ControlButtonArrived` Event” on page 196.

- ◆ For the `EVENTNUM_MouseMoved` event; 1 means the button is down, and 0 means the button is up.
- ◆ `med_HorizPosition` and `med_VertPosition` report the mouse’s current position in absolute space. That position is reckoned from an origin (0,0) that is set when the mouse is first plugged in, and uses units that will typically (depending on the mouse) measure 100, 200, or 400 increments per inch. It is up to the task to interpret the absolute position of the mouse to a pointer position on the display.

Light Gun Data Structure

The data structure that reports light gun data is `LightGunData`, defined in `event.h` as follows:

Example 13-5 `LightGunEventData` structure

```
typedef struct LightGunEventData
{
    uint32  lged_ButtonBits;      /* left justified, zero fill */
    uint32  lged_Counter;         /* counter at top-center of hit */
    uint32  lged_LinePulseCount; /* # of scan lines that were hit */
} LightGunEventData;
```

- ◆ `lged_ButtonBits` contains a value whose bits tell the state of the various triggers, buttons, and other pushable or flippable controls on a light gun. Most light guns have one trigger and one auxiliary button. The only flag bit currently defined for this field is:
 - ◆ `LightGunTrigger` - the main light gun trigger.
- ◆ `lged_Counter` contains a value which specifies the number of times that the light gun’s internal clock (nominally 20 MHz) counted up, between the time it was reset (during vertical blanking) and the time that the light gun’s optical sensor “saw” a flash of light from the display as the display’s electron beam passed through its field of view. The beam travels left to right along each line, and downwards from line to line.
- ◆ `lged_LinePulseCount` contains the value 0 if the light gun sensor did not detect a sufficiently strong pulse of light while scanning the video field. In this case, the `lged_Counter` value may or may not be valid. If the light gun sensor detected a sufficiently strong pulse of light, `lged_LinePulseCount` contains a nonzero value. Some light guns can actually count the number of successive horizontal scan lines during a pulse and return that value in this

field. For light guns of this sort, the `lged_LinePulseCount` is considered a "quality of signal" indicator. Other light guns simply return a constant value (typically 15) to indicate that their sensor detected at least one pulse that was strong enough to trip the sensor.

Joystick Data

The data structure used to report joystick data is `StickEventData`, defined in `event.h` as follows:

Example 13-6 `StickEventData` structure

```
typedef struct StickEventData
{
    uint32      stk_ButtonBits;      /* left justified, zero fill */
    int32       tk_HorizPosition;
    int32       stk_VertPosition;
    int32       tk_DepthPosition;
} StickEventData;
```

- ◆ `stk_ButtonBits` contains a value that identifies which joystick button (or buttons) generated the event. The constants defining these flag bits are as follows:
 - ◆ `StickCapability` — the AND of `Stick4Way` and `StickTurbulence`.
 - ◆ `Stick4Way` — determines how many buttons the stick has.
 - ◆ `StickTurbulence` — indicates whether or not the stick understands output commands.
 - ◆ `StickButtons` — the stick buttons.
 - ◆ `StickFire` — the joystick was fired.
 - ◆ `StickA` — the A button.
 - ◆ `StickB` — the B button.
 - ◆ `StickC` — the C button.
 - ◆ `StickUp` — the up button of the stick.
 - ◆ `StickDown` — the down button of the stick.
 - ◆ `StickRight` — the right button of the stick.
 - ◆ `StickLeft` — the left button of the stick.
 - ◆ `StickPlay` — the play button of the stick.
 - ◆ `StickStop` — the stop button of the stick.
 - ◆ `StickLeftShift` — the left-shift button of the stick.
 - ◆ `StickRightShift` — the right-shift button of the stick.
- ◆ `stk_HorizPosition`, `stk_VertPosition`, and `stk_DepthPosition` contain binary numbers and return a value between 0 through 1023. A value

of 0 means the joystick is pushed left or all of the way down. A value of 1023 means the joystick is pushed right or all the way up. Keep in mind that some joysticks cannot go all the way to 0 or 1023.

File System State Data Structure

When a filesystem is mounted, dismounted, or placed off-line (EVENTNUM_FileSystemMounted, EVENTNUM_FileSystemDismounted, or EVENTNUM_FileSystemOffline), the FileSystemEventData structure contains the item number of the filesystem node, and the name of the filesystem which is changing state. The FileSystemEventData structure is defined as follows:

Example 13-7 FileSystemEventData structure

```
typedef struct FileSystemEventData
{
    Item    fsed_FileSystemItem;
    char    fsed_Name[FILESYSTEM_MAX_NAME_LEN];
} FileSystemEventData;
```

High-Performance Event Broker Use

The pod table interface gives applications a fast and efficient way to monitor the current state of certain control port devices on a field-by-field basis. The pod table does not require applications to process a large number of event messages.

The Pod Table

Devices such as analog joysticks and light guns provide applications with position information, as well as with button-state information. Button-state information does not change frequently (a few times per second in most cases), but positional information changes often during almost every video field (up to 60 times a second). These rapid position changes are inherent in the high resolutions of these input devices, and their tendency to “jitter” slightly as a result of small mechanical movements or electrical noise in the circuitry. Applications must track the position of an input device. For example, by moving a set of crosshairs on the display, an application may need to parse and process messages from the event broker many times each second. This process can lead to an undesirable slowdown of the application.

The *pod table* is an alternative method for accessing the current position of devices such as the analog joystick and light gun, with substantially lower overhead for both the application and the event broker. This table (actually a set of arrays and

data structures) contains the current position information, and current button-state, for up to eight generic control port devices in each family. By examining the contents of the pod table, applications can track device position whenever it is convenient for them to do so.

Gaining Access to the Pod Table

The event broker constructs the pod table when an application sends it a message asking for access to the table. This message has the standard event broker format and uses a message code of `EB_MakeTable`.

The event broker builds the table and replies to the application's message by sending an `EB_MakeTableReply` message whose data structure includes a pointer to the table. The table is updated whenever new data arrives from the control port. An application that gains access to the table in this fashion can "poll" the contents of the table at its convenience. The table contains a semaphore that the application must lock to ensure the event broker does not update the table while the application examines it.

Relinquishing Access to the Pod Table

The pod table is retained in memory and kept up to date, as long as the application that requested it is executing. The system maintains the table as long as the event broker receives a table-request message from at least one message port. As soon as the event broker detects that the last such message port was deleted, it ceases updating the pod table and releases the memory occupied by the table.

Structure of the Pod Table

The pod table consists of two portions: a fixed-size and fixed-format section and a set of variable-size arrays, which are created when needed. The variable size arrays may not always be present. Figure 13-1 shows the fixed-size section and fixed-format.

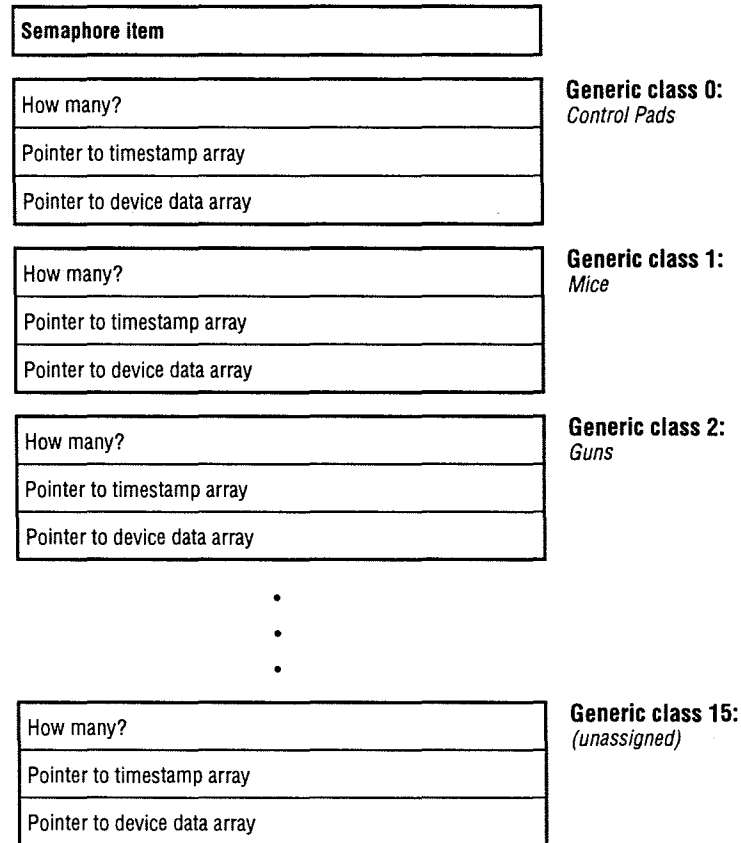


Figure 13-1 Pod table structure fixed-size and fixed-format section.

The table begins with a semaphore item. It is followed by a total of 16 almost-identical substructures: one substructure per generic class, with the names of the substructures indicating the generic class to which they refer. Within each substructure are two pointers: one to an array of `uint32` timestamps, and another to an array of generic class-specific event data structures. Each substructure also contains a `uint32` “how many” field that gives the number of elements in the event and timestamp arrays. The “how many” field can contain 0, in which case the two pointers will contain NULL.

The event data structures used in these arrays are the same ones used to send information in a normal event broker event frame. For example, the substructure for generic class 0 (control pads) contains a pointer to an array of `ControlPadEventData` structures, and the substructure for generic class 8 (analog joysticks) contains a pointer to an array of `StickEventData` structures.

Empty Generic Class Substructures

If there are no devices of a particular generic class connected to the 3DO unit, the substructure for this generic class can be left empty. In this case, the “how many” field contains 0, and the two array pointers will contain NULL. For example, if the 3DO unit were started up without a control pad attached, the first portion of the pod table might be as shown in Figure 13-2.

Semaphore Item	
How many?	0
Pointer to timestamp array	(null)
Pointer to device data array	(null)

•
•
•

Generic class 0:
Control Pads

Figure 13-2 Pod substructure.

Non-Empty Generic Class Substructures

If one or more devices of a particular generic class are connected to the 3DO system, the pod table reflects this. The “how many” field is nonzero, and the array pointers are non-NULL and point to valid arrays. Figure 13-3 shows an example of the generic class substructure.

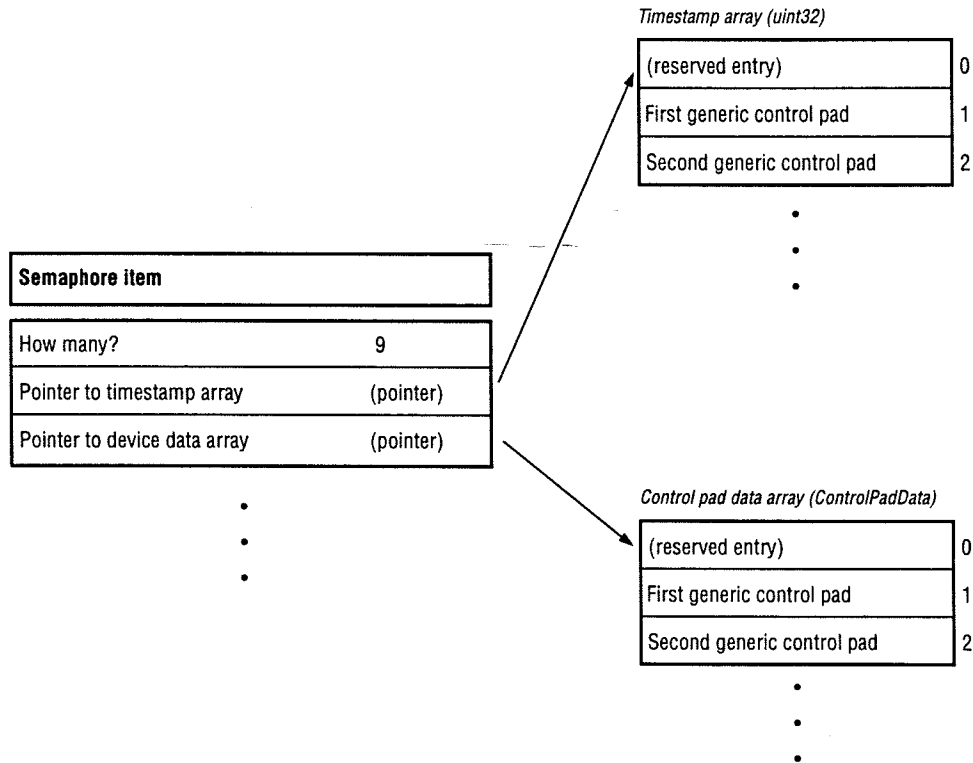


Figure 13-3 Generic class substructure.

The “how many” field does not necessarily contain the number of devices of a generic class that are currently attached to the system; rather, this field identifies the sizes of the data arrays. Normally, the event broker reserves sufficient space in the arrays for up to eight devices in each class, and the “how many” field will reflect this fact. If your application needs to get a complete and accurate listing of the devices currently attached to the system, it should use the `EB_DescribePods` message to ask the event broker for a current list.

The entries in the timestamp and device-data arrays should be accessed with a starting index of 1. That is, the data for the first generic joystick will be found in `StickEventData[1]`. This ensures that the numbering scheme for these arrays matches the generic-device numbers in the event broker’s event-frame and pod-description messages. Entry 0 in each array is reserved for future use, and should not be accessed by applications.

The values in the timestamp arrays are standard Portfolio VBL counter timestamps. They have a resolution of 1/60th of a second on NTSC systems and 1/50th of a second on PAL systems. They are derived from the same source as the event timestamp values in an event broker event message, and also correspond to the VBL-count times applications read using the timer device.

If a controller is unplugged from the control port, the event broker ceases updating its entries in the pod table. An application can detect stale information by checking to see if the validity timestamp is changing.

A substructure goes from empty to nonempty if a device of an as yet unseen generic class is connected to the multiplayer, and a driver for this device is available. The array pointers in a substructure can be changed if the event broker needs to enlarge the arrays to support a larger number of devices than expected. In any case, a change in the table's structure will occur only when the event broker has locked the semaphore on the pod table. Applications are urged to lock the semaphore before accessing the contents of the table, and unlock it immediately afterwards, to avoid having data in the table change while the application is reading it.

Use and Abuse of the Pod Table

It is easy to misuse the pod table, which can hurt the performance and reliability of your application, rather than helping it.

The pod table mechanism is intended to give applications a way to view the most recent information for each control port device on the system. As such, it lets you track a light gun's target position or the offset of an analog joystick with very little effort and overhead. You may be tempted to use it for other, less appropriate purposes: for example, to detect button-down and button-up interactions at the control pads or light gun or analog joysticks. *Do not do this.* It can cause compatibility problems, now or in the future.

You can also get into trouble because an application has no guaranteed way of knowing just when the data in the pod table is likely to be changed. If somebody taps and releases a button on the control pad very quickly, it is possible for the pod table to be updated twice (reflecting both the button-down and button-up changes) before your application gets around to polling it. As a result, your application will never notice this particular button-press, and it will not react to the user's action.

Pod Table Polling

You may think that you can avoid this problem by polling the pod table frequently, perhaps once per field (60 times a second on an NTSC machine), but this is not recommended. The 3DO hardware and software will probably scan the control port more frequently than once per field in order to reduce the delay

between the time a button is pressed and the time a program reacts to it. If your application depends on polling the pod table more rapidly than the event broker can update it, your application will probably not run reliably.

Polling the pod table frequently adds quite a bit of overhead and complexity to an application. You will either have to interrupt your main-line code several times per field to poll the table (making the code bigger and bulkier) or it will be necessary to have a code thread wake up every few milliseconds to check the table (thus adding more overhead than the pod table mechanism eliminated, and slowing down your application's frame rate). You might be tempted to start writing "busy/wait" polling loops to wait for events; these would make your program multitasking-hostile and network-unfriendly.

The event broker's event message system was designed specifically to eliminate such problems. The event broker is a couple of steps closer to the hardware than your application is, and takes advantage of this fact. It does all of the button-down/button-up detection for you. Every time new data arrives from the control port it can queue up several event messages in a row for your application without losing any data, and it will wake up your program or thread (sending a message) when an event arrives.

The best way to use the pod table is to use it *only* to keep track of position information (analog joystick-position, mouse offset, light gun target pulse count, and so on.). For all real user-interaction events (button-down, trigger pulled, and so on), use the event broker event message mechanism.

Using the Pod Table Semaphore

When you lock the semaphore on the pod table before accessing the table, be sure to unlock it as quickly as possible. If it's left locked, you will lock out the event broker and can cause your application to miss events. Ideally, you should lock the semaphore, copy the information you need from the pod table to a variable in your program's own storage, and then unlock the semaphore immediately.

It is tempting to not lock the semaphore and read the tables directly. This is dangerous, as you might get inconsistent data within the table. It might also lead to crashes if new controllers are plugged into the system and the event broker must alter table pointers.

The semaphore should be locked in read mode using the `SEM_SHAREDREAD` flag.

Finally, since the array pointers within the pod table can change from event to event, it is important not to cache these values anywhere. Whenever you lock the pod table semaphore, you must sample the array pointers from the table.

Synchronization Between the Pod Table and Event Messages

The event frames in an event broker message have timestamps associated with each event, as do the arrays in the pod table. Both timestamps are derived from the same source (the timer device VBL counter) and so are consistent with one another.

However, if you are checking timestamps in the event messages, and are also looking at timestamps in the pod table, you should be cautious. It is possible for you to receive an event message (containing a timestamped event), look in the pod table, and find that the timestamp for the corresponding device does not match the one in the event message. This can (and often will) happen because the pod table was updated between the time that the event message was sent to your application, and the time that your application processed it. The pod table timestamp will usually be more recent than the event frame timestamp.

This might cause some confusion if, for example, an application tracking the light gun cursor based on pod table information but the application was “firing bullets” based on the event messages. The application might mistakenly shoot a bullet at the position the gun was aimed before or after the moment the trigger was pulled, rather than where it was aimed at the precise moment of firing.

All of the information in any particular event frame has the same timestamp. For example, if your application receives a “light gun button down” event frame, then the button-bits and the light gun counter in that event frame were sampled at exactly the same moment. Your application should fire at the spot indicated in the event frame, rather than at the (possibly different) location currently indicated in the pod table.

Reconfiguring or Disconnecting a Task

As a task works with the event broker, it may want to reconfigure itself to monitor different event types, change its focus dependence, reset the size of its event queue, or use a new reply port for event broker messages. Or it may want to completely disconnect itself from the event broker so that it no longer monitors events. In each case, the task sends a new configuration message to the event broker.

Reconfiguring the Event Broker Connection

The task initializes a new `ConfigurationRequest` structure with the new settings it wants. It then sends an `EB_Configure` message to the event broker to have the new settings take effect. The event broker uses the reply port of the `EB_Configure` message to determine which tasks are connected to the event broker. When changing the configuration, the application should use the same reply port for the message as it did when it first connected to the event broker.

If a task wants to change the reply port it uses to receive event broker messages, it must first disconnect from the event broker using a configuration message with its old reply port. Then the task must create a new configuration message with a new reply port, and send the new configuration message to the event broker. The task must also close the old configuration message and old reply port.

Disconnecting From the Event Broker

If the task wishes to disconnect itself completely from the event broker, it sends a configuration request, using its current reply port, and specifies a 0 for both the trigger and capture masks. The event broker then disconnects the task. A task can also call `DeleteMsgPort()` on its current reply port. On the next tick, the event broker will disconnect the task.

Other Event Broker Activities

Although the event broker's primary activity is monitoring pod table events for listeners, it performs other activities such as checking on connected listeners and pods, finding out where the focus lies, and controlling individual pods, such as stereoscopic glasses and audio controllers.

Getting Lists of Listeners and Connected Pods

There are many times when a task needs a list of pods connected to the control port or listeners connected to the event broker. There are two flavors of messages that retrieve this kind of information: `EB_DescribePods` asks for a list of connected pods; `EB_GetListeners` asks for a list of connected listeners.

Asking for a List of Connected Pods

When a task uses an `EB_DescribePods` message to ask the event broker for a list of connected pods, the message data block contains only the `EventBrokerHeader` data structure specifying the flavor `EB_DescribePods`. The message sent to the event broker should be a buffered message, created with `CreateBufferedMsg()`, which contains sufficient room to hold the complete list of connected pods.

Reading a Returned List of Connected Pods

When the event broker replies to an `EB_DescribePods` message, it turns the message into an `EB_DescribePodsReply` message. The message's data buffer contains a list of pods currently attached to the control port. Those pods are described in the data structure `PodDescriptionList`, which is defined as follows:

Example 13-8 PodDescriptionList structure

```
typedef struct      PodDescriptionList
{
    EventBrokerHeader    pdl_Header;
    int32                pdl_PodCount;
    PodDescription       pdl_Pod[1];
} PodDescriptionList;
```

- ◆ pdl_Header is an EventBrokerHeader data structure set to the message flavor EB_DescribePodsReply.
- ◆ pdl_PodCount contains the number of pods described in the next field.
- ◆ pdl_Pod[] is an array of pod descriptions, with one element for each pod currently attached to the control port. Each pod description is contained in the data structure PodDescription, defined as follows:

Example 13-9 PodDescription structure

```
typedef struct PodDescription
{
    uint8        pod_Number;
    uint8        pod_Position;
    uint8        rfu[2];
    uint32       pod_Type;
    uint32       pod_BitsIn;
    uint32       pod_BitsOut;
    uint32       pod_Flags;
    uint8        pod_GenericNumber[16];
    Item         pod_LockHolder;
} PodDescription;
```

- ◆ pod_Number gives the unique and unchanging pod number of the pod.
- ◆ pod_Position gives the position of the pod in the control port daisy chain. It is an integer from 1 to the total number of pods on the control port.
- ◆ pod_Type lists the native pod type of the pod, that is, the specific type of device attached to the daisy chain (not the generic type).
- ◆ pod_BitsIn and pod_BitsOut give the number of bits shifted into the pod during the control port data stream for each video field, and the number of bits shifted out of the pod during each data stream.

- ◆ `pod_Flags`, contains flags that show the generic device type (or types) that the pod contains. These flags, shown in Table 13-4, occupy the left-most bits of the `pod_Flags` word.

Table 13-4 *Flag constants used for the `pod_Flags` field.*

Flag Constant	Hex Value	Count From Left-Most Bit
<code>POD_IsControlPad</code>	<code>0x80000000</code>	0
<code>POD_IsMouse</code>	<code>0x40000000</code>	1
<code>POD_IsGun</code>	<code>0x20000000</code>	2
<code>POD_IsGlassesCtrlr</code>	<code>0x10000000</code>	3
<code>POD_IsAudioCtrlr</code>	<code>0x08000000</code>	4
<code>POD_IsKeyboard</code>	<code>0x04000000</code>	5
<code>POD_IsLightGun</code>	<code>0x02000000</code>	6
<code>POD_IsStick</code>	<code>0x01000000</code>	7
<code>POD_IsIRController</code>	<code>0x00800000</code>	8

Note: *Multiple bits in this flag field may be set (e.g., `POD_IsControlPad` and `POD_IsAudioCtrlr`). Furthermore, bits may be set that are undocumented because they are for internal use only. When using the `pod_Flags` flag word, check for the status of individual bits. Do NOT compare the whole word to an overall value.*

- ◆ `pod_GenericNumber` is an array of 16 unsigned 8-bit values. These values correspond to the bits of the `pod_Flags` field: `pod_GenericNumber[0]` corresponds to flag bit 0, `POD_IsControlPad`; `pod_GenericNumber[1]` corresponds to the first flag bit, `POD_IsMouse`; and so on. The value in each element of the array stores the rank of this pod in the order of all other identical generic devices connected to the serial port. For example, if `pod_GenericNumber[2]` contains a 4, then the pod contains a generic gun that is the fourth generic gun connected to the serial port.
- ◆ `pod_LockHolder` gives the item number of the task that has this pod locked for its exclusive use. If this field is set to 0, then the pod is unlocked. Pod locking is not currently implemented, so this field should not be used.

Asking for a List of Connected Listeners

When a task asks for a list of connected listeners, it uses an `EB_GetListeners` message, whose data block contains only the `EventBrokerHeader` data structure specifying the flavor `EB_GetListeners`. Messages used for this purpose must be created with `CreateBufferedMsg()` in the same way as messages used with the `EB_DescribePods` command. `CreateBufferedMsg()` is described in “Creating a Configuration Message” on page 183.

Reading a Returned List of Connected Listeners

When the event broker replies to an `EB_GetListeners` message, it turns the message into an `EB_GetListenersReply` message. The message buffer contains a list of listeners currently connected to the event broker. The list is contained in the data structure `ListenerList`, defined as follows:

Example 13-10 *ListenerList structure*

```
typedef struct ListenerList
{
    EventBrokerHeader    ll_Header; /* { EB_GetListenersReply } */
    int32                ll_Count;
    struct {
        Item             li_PortItem;
        enum ListenerCategory li_Category;
    } ll_Listener[1];
} ListenerList;
```

- ◆ `ll_Header` is an `EventBrokerHeader` structure set to the message flavor `EB_GetListenersReply`.
- ◆ `ll_Count` contains the number of listeners described in the next field.
- ◆ `ll_Listener` is an array of listener descriptions with one element for each listener currently connected to the event broker. Each listener element contains these fields:
 - ◆ `li_PortItem` contains the item number of the listener's reply port
 - ◆ `li_Category` gives the listener's focus-interest category

Working With Input Focus

It is often important for a listener to know which task currently has the input focus or to be able to change the input focus from one task to another without user intervention. The event broker can handle both of these requests.

Finding the Current Focus Holder

To find the current focus holder, a listener can inquire with an `EB_GetFocus` message. Its data block is an `EventBrokerHeader` data structure set to the message flavor `EB_GetFocus`.

When the event broker receives an `EB_GetFocus` message, it gets the item number of the task currently holding the focus, writes that value in the error field of the message, and returns the message to the task. The task receiving the message reads the current focus holder's item number from the error field. The error field will contain a negative number (an error code) if the operation failed.

Setting the Current Focus Holder

To move the focus from one task to another without input from the user, a listener can use an `EB_SetFocus` message. The data block for this message uses the `SetFocus` data structure, defined as follows:

Example 13-11 *SetFocus structure*

```
typedef struct SetFocus
{
    EventBrokerHeader sf_Header;          /* { EB_SetFocus } */
    Item              sf_DesiredFocusListener;
} SetFocus;
```

- ◆ `sf_Header` is an `EventBrokerHeader` data structure set to the message flavor `EB_SetFocus`.
- ◆ `sf_DesiredFocusListener` is the item number of the task to which the event broker should give the focus.

When the event broker receives the message, it changes the input focus to the requested task and writes that task's item number into the error field of the message. It returns the message to the sending task. That task can now read the error field to get the item number of the focus-holding task (if successful) or an error code (if unsuccessful).

Commanding a Pod

Sometimes a task may want to control a pod through the event broker. For example, a task may wish to alternate lens opaqueness in a pair of stereoscopic glasses, or mute the sound coming through an audio controller. To issue commands to a pod, the task uses a message of the flavor `EB_IssuePodCmd`. It accompanies the message with a data block that uses the `PodData` data structure, defined as follows:

Example 13-12 PodData structure

```
typedef struct PodData
{
    EventBrokerHeader    pd_Header;
    int32                pd_PodNumber;
    int32                pd_WaitFlag;
    int32                pd_DataByteCount;
    uint8                pd_Data[4];
} PodData;
```

- ◆ `pd_Header` is an `EventBrokerHeader` data structure set to the message flavor `EB_IssuePodCmd`.
- ◆ `pd_PodNumber` gives the pod number of the pod to which the command is sent.
- ◆ `pd_WaitFlag` can be set to either 1 or 0. If set to 1, it asks the event broker to send the command to the pod and reply immediately. If set to 0, it asks the event broker to send the command to the pod, wait for the command to finish execution, and then reply to command message.
- ◆ `pd_DataByteCount` gives the size in bytes of the `pd_Data` field that follows.
- ◆ `pd_Data` is an array of bytes that contains the command sent to the pod. The first element of the array (element 0) contains the generic device class of the pod to which the command is sent; the second element (element 1) contains the command subcode, which is specific to the generic class in the first element. If the command defined by these two bytes requires extra data, that data goes into the elements of the array listed below.

Generic Device Class

The include file *event.h* currently defines the following constants to specify generic device classes:

- ◆ `GENERIC_ControlPad` – a control pad.
- ◆ `GENERIC_Mouse` – a mouse.
- ◆ `GENERIC_Gun` – a photo-optic gun.
- ◆ `GENERIC_GlassesCtrlr` – a stereoscopic glasses controller.
- ◆ `GENERIC_AudioCtrlr` – an audio controller.
- ◆ `GENERIC_Keyboard` – a keyboard.
- ◆ `GENERIC_LightGun` – a light gun.
- ◆ `GENERIC_Stick` – a joystick.
- ◆ `GENERIC_IRController` – a infrared control pad (wireless).

The array element `pd_Data[0]` should hold the constant appropriate to the commanded pod.

Audio Controller Subcodes

The audio controller accepts commands defined by audio controller subcodes. At present, Portfolio offers one audio subcode, specified by the following constant:

- ◆ `GENERIC_AUDIO_SetChannels` controls the output of the two stereo audio channels.

This subcode goes in the array element `pd_Data[1]`. It requires one byte of data following it in `pd_Data[2]`. This data specifies how the stereo channels are controlled. The four options, defined by the following constants, are:

- ◆ `AUDIO_Channels_Mute` – turns off the audio output in both channels.
- ◆ `AUDIO_Channels_RightToBoth` – feeds the right audio signal to both left and right channels.
- ◆ `AUDIO_Channels_LeftToBoth` – feeds the left audio signal to both the left and right channels.
- ◆ `AUDIO_Channels_Stereo` – feeds the left audio signal to the left channel and right audio signal to the right channel.

Glasses Controller Subcodes

The stereoscopic glasses controller accepts commands defined by glasses controller subcodes. At present, Portfolio offers one glasses subcode, specified by the following constant:

- ◆ `GENERIC_GLASSES_SetView` – controls the opacity of each lens in a pair of stereoscopic glasses.

This subcode goes into the array element `pd_Data[1]`. It requires two bytes of data following it in `pd_Data[2]` and `pd_Data[3]`. The first of the two data bytes controls the left lens of the glasses, the second byte controls the right lens. The four possible values for each lens, defined by the following constants, are:

- ◆ `GLASSES_AlwaysOn` – keeps the lens clear during both odd and even video fields.
- ◆ `GLASSES_OnOddField` – turns the lens clear every odd video field, and turns it opaque every even field.
- ◆ `GLASSES_OnEvenField` – turns the lens clear every even video field, and turns it opaque every odd field.
- ◆ `GLASSES_Opaque` – turns the lens opaque during both odd and even video fields.

Light Gun Controller Subcode

Currently, Portfolio offers one light gun controller subcode, which is specified by the following constant:

- ◆ `GENERIC_LIGHTGUN_SetLEDs` controls the setting of LEDs on the light gun. The setting of up to eight LEDs is supported, although the stock light gun has only two.

The possible options, defined by the following constants, are:

- ◆ `LIGHTGUN_LED1` – controls the setting of the first LED on the light gun.
- ◆ `LIGHTGUN_LED2` – controls the setting of the second LED on the light gun.

The Event Broker's Reply to a Command

After the event broker processes a pod command message, it returns the message to the sending task to report on the command execution. The returned message is of the flavor `EB_IssuePodCmdReply`, and contains a reply portion that is a negative number if there was an error executing the command, or is 0 or a positive number if the command execution was successful.

Event Broker Convenience Calls

Portfolio provides a set of convenience calls that allow a task to easily monitor events on control pads or mice.

Note: *The convenience calls are "single threaded." Only one thread within any application can use them at a time.*

Connecting to the Event Broker

This call connects the task to the event broker, sets the task's focus interest, and determines how many control pads and mice the task wants to monitor:

```
Err InitEventUtility(int32 numControlPads, int32 numMice,  
                    int32 focusListener);
```

Where:

- ◆ `numControlPads` sets the maximum number of control pads the task wants to monitor.
- ◆ `numMice` sets the maximum number of mice the task wants to monitor. These values should be an integer of 0 or higher.
- ◆ `focusListener` is a Boolean value that sets the focus of the task when it is connected as a listener. If the parameter is true (nonzero), the task is connected as a focus-dependent listener. If the parameter is false (0), the task is connected as a focus-independent listener.

When the call executes, it creates a reply port and a message, sends a configuration message to the event broker that asks the event broker to report appropriate mouse and control pad events, and deals with the event broker's configuration reply. The call returns 0 if successful, and a negative number (an error code) if unsuccessful.

Monitoring a Control Pad or a Mouse

This call specifies a control pad or mouse to monitor, specifies whether the event broker should respond immediately or wait until something happens on the control pad, and provides a data structure for data from the control pad:

```
Err GetControlPad(int32 padNumber, int32 wait,
                  ControlPadEventData *data);
```

- ◆ `padNumber` sets the number of the generic control pad on the control port (i.e., the first, second, third, and so on, pad in the control port daisy chain, counting out from the 3DO unit) that the task wants to monitor. The first pad is 1, the second is 2, and so on.
- ◆ `wait` is a Boolean value that specifies the event broker's response to this call. If it is true (nonzero), the event broker waits along with the task until an event occurs on the specified pad, and returns data only when there is a change in the pad. If it is false (0), the event broker immediately returns with the status of the pad.
- ◆ `data` is a pointer to a `ControlPadEventData` structure that shows where memory was allocated for the returned control pad data to be stored. (The task must create an instance of this data structure before it executes `GetControlPad()`).

When the call executes, it either returns immediately with event information, or waits to return with event information until there is a change in the control pad. It returns a 1 if an event has occurred on the pad, a 0 if no event has occurred on the pad, or a negative number (an error code) if there was a problem retrieving an event. If an event has occurred, the task must check the `ControlPadEventData` data structure for details about the event.

This call performs the same function as `GetControlPad()`, but gets events from a specified mouse instead of a specified control pad:

```
Err GetMouse(int32 mouseNumber, int32 wait, MouseEventData *data)
```

Its parameters are the same as those for `GetControlPad()`, but `mouseNumber` specifies a mouse on the control port instead of a control pad, and the pointer data is now a pointer to a data structure for storing mouse event data instead of control pad event data.

Disconnecting From the Event Broker

This call disconnects a task that was connected to the event broker with the `InitEventUtility()` call:

```
Err KillEventUtility(void)
```

When executed, it disconnects the task from the event broker, closes the reply port, and frees all resources used for the connection. It returns 0 if successful, and a negative number (error code) if unsuccessful.

The International Folio

This chapter describes the International and the JString folios for use in creating titles suitable for international markets.

This chapter contains the following topics:

Topic	Page Number
What Is Internationalization?	217
The Locale Data Structure	218
Using the Locale Structure	223

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

What Is Internationalization?

Internationalization is the process by which software is developed and modified so it transparently adapts to multiple cultural environments. *Localization* is the process of preparing specialized software, or special versions of existing software, targeted to individual cultural environments.

Internationalization of software has the following advantages:

- ◆ It lets you maintain a single set of source code for each title, instead of slightly different source code for each target culture. A single set of sources considerably reduces the testing burden and increases confidence in product reliability.

- ◆ It potentially lets both you and The 3DO Company deal with fewer master CD-ROMs than would otherwise be required. Instead of producing a different CD-ROM for each target culture, a single CD-ROM can support all cultures.

The International folio provides a structure and calls that let you create a title for international environments. For example, using the Folio, your application can determine the current language and country codes, display dates, currency, and numeric values that are consistent with the current language and country codes of a target culture. The International folio also works with a separate folio, JString, to convert character sets from one format to another.

The International folio uses UniCode as its base character set. The functions and structures within the folio use the `unichar` data type, which is a 16-bit version of the standard `char` type.

UniCode is a standard defining a 16-bit character set, which is essentially an international version of ASCII. UniCode specifies enough glyphs to represent all languages currently spoken around the world.

Although UniCode is the standard character set the International folio uses, it must support other character sets. For example, the Western computer world currently relies extensively on ASCII, and Japan relies on JIS and shift JIS. The International folio provides tools to convert strings from one character set to another, such as enabling applications to convert ASCII text into UniCode. The following books provide additional information on the UniCode standard:

- ◆ *The UniCode Standard Worldwide Character Encoding Version 1.0, Volume 1*
ISBN 0-201-56788-1
- ◆ *The UniCode Standard Worldwide Character Encoding Version 1.0, Volume 2*
ISBN 0-201-60845-6

The Locale Data Structure

The `Locale` data structure is like a little database the system provides. Once you get a pointer to the `Locale` structure, it has, among other things, the current language being used, the country code, and the format of dates for the user's cultural environment.

The system creates a `Locale` structure when an application calls `intlOpenLocale()`. `intlOpenLocale()` returns an `Item`, which must be disposed of using the `intlCloseLocale()` macro.

To examine the contents of the `Locale` structure, an application calls `intlLookupLocale()`, which returns a pointer to the `Locale` structure for the specified `Item`. The definition of the `Locale` structure is shown below.

Example 14-1 Locale structure

```

typedef struct Locale
{
    ItemNode          loc_Item;                /* system linkage */

    /* preferred language to use */
    LanguageCodes     loc_Language;

    /* An array of dialects for the current language, listed in order
       * of preference, and terminated with INTL_DIALECTS_ARRAY_END
       */
    DialectCodes      *loc_Dialects;

    /* ISO-3166 numeric-3 code for the user's country */
    CountryCodes      loc_Country;

    /* general description of the user's environment */
    int32             loc_GMTOffset;           /* minutes from GMT */
    MeasuringSystems   loc_MeasuringSystem;    /* measuring system to use */
    CalendarTypes      loc_CalendarType;      /* calendar type to use */
    DrivingTypes       loc_DrivingType;       /* side of the street */

    /* number formatting */
    NumericSpec        loc_Numbers;
    NumericSpec        loc_Currency;
    NumericSpec        loc_SmallCurrency;

    /* date formatting */
    DateSpec           loc_Date;
    DateSpec           loc_ShortDate;
    DateSpec           loc_Time;
    DateSpec           loc_ShortTime;
} Locale;

```

The fields in the Locale structure are as follows:

- ◆ loc_Item provides system linkage for Locale structures which are allocated in system space so that they can be shared among multiple applications.
- ◆ loc_Language defines the language to use within an application. Each supported language has a code, which is taken from the ISO 639 Standard.

- ◆ `loc_Dialects` is a pointer to an array of dialects. Regional variations within a given language are accommodated through dialect tables, which are an array of dialect codes, terminated by `INTL_DIALECT_ARRAY_END`.

The dialects appear in the array in decreasing order of user preference. For example, if the language is `INTL_LANG_ENGLISH`, then the dialect array could hold `INTL_ED_AMERICAN`, `INTL_ED_AUSTRALIAN`, `INTL_ED_BRITISH`, and so on.
- ◆ `loc_Country` contains the standard international country code for the current country. These codes are taken from the ISO 3166 Standard.
- ◆ `loc_GMTOffset` contains the offset in minutes of the current location relative to the standard GMT reference point.
- ◆ `loc_MeasuringSystem` indicates the measuring system to use. This can be `INTL_MS_METRIC`, `INTL_MS_AMERICAN`, or `INTL_MS_IMPERIAL`.
- ◆ `loc_CalendarType` indicates what type of calendar to use. This can be the traditional Gregorian calendar, with either Monday or Sunday as the first day of the week, or it can be the Arabic, Jewish, or Persian calendar.
- ◆ `loc_DrivingType` indicates on which side of the street cars usually travel in the current country.
- ◆ `loc_Numbers`, `loc_Currency`, `loc_SmallCurrency` specifies how to format numbers and currency. The `NumericSpec` structure contains the necessary information to properly localize number output and input. These three `NumericSpec` structures can be passed directly to the `intlFormatNumber()` function to apply localized formatting.
- ◆ `loc_Date`, `loc_ShortDate`, `loc_Time`, `loc_ShortTime` specifies how to format dates and time. The `DateSpec` array contains the information needed to localize date and time output and input properly. These four `DateSpec` arrays can be passed directly to the function `intlFormatDate()` to apply localized formatting.

NumericSpec Structure

Each culture has its own way of writing numbers and currency. The `Locale` structure contains three `NumericSpec` structures that contain number and currency formatting specifications and are used to produce correctly localized output for the target cultures.

The `NumericSpec` structure defines how numbers should be formatted. It is sufficiently flexible to cover plain numbers and currency values. The structure is shown below.

Example 14-2 NumericSpec structure

```

typedef struct NumericSpec
{
    /* how to generate a positive number */
    GroupingDesc    ns_PosGroups;           /* grouping description */
    unichar          *ns_PosGroupSep;       /* separates the groups */
    unichar          *ns_PosRadix;          /* decimal mark */
    GroupingDesc    ns_PosFractionalGroups; /* grouping description */
    unichar          *ns_PosFractionalGroupSep; /* separates the groups */
    unichar          *ns_PosFormat;         /* for post-processing */
    uint32           ns_PosMinFractionalDigits; /* min # of frac digits */
    uint32           ns_PosMaxFractionalDigits; /* max # of frac digits */

    /* how to generate a negative number */
    GroupingDesc    ns_NegGroups;           /* grouping description */
    unichar          *ns_NegGroupSep;       /* separates the groups */
    unichar          *ns_NegRadix;          /* decimal mark */
    GroupingDesc    ns_NegFractionalGroups; /* grouping description */
    unichar          *ns_NegFractionalGroupSep; /* separates the groups */
    unichar          *ns_NegFormat;         /* for post-processing */
    uint32           ns_NegMinFractionalDigits; /* min # of frac digits */
    uint32           ns_NegMaxFractionalDigits; /* max # of frac digits */

    /* when the number is zero, this string is used 'as-is' */
    unichar          *ns_Zero;
} NumericSpec;

```

Using the fields in the NumericSpec structure, the `intlFormatNumber()` function can correctly format numbers and currency according to local rules and customs. The fields of the NumericSpec structure are as follows:

- ◆ `ns_PosGroups` defines how digits are grouped left of the decimal mark. A `GroupingDesc` is a 32-bit bitfield in which every ON bit in the bitfield indicates a separator sequence should be inserted after the associated digit. If bit 0 is ON, the grouping separator is inserted after digit 0 of the formatted number.
- ◆ `ns_PosGroupSep` is a string that separates groups of digits to the left of the decimal mark.
- ◆ `ns_PosRadix` is a string used as a decimal character.
- ◆ `ns_PosFractionalGroups` is a `GroupingDesc` array that defines how digits are grouped to the right of the decimal character.

- ◆ `ns_PosFractionalGroupSep` is a string used to separate groups of digits to the right of the decimal mark.
- ◆ `ns_PosFormat` is used for post-processing on a formatted number. Typically it is used to add currency notation around a numeric value. The string in this field is used as a format string in `sprintf()`, and the formatted numeric value is also a parameter to `sprintf()`.

For example, if `ns_PosFormat` is defined as `$$s`, and the formatted numeric value is 0.02. The result of the post-processing will be \$0.02. When this field is NULL, no post-processing occurs.

- ◆ `ns_PosMinFractionalDigits` specifies the minimum number of characters to the right of the decimal mark. If there are fewer characters than the minimum, the number is padded with 0s.
- ◆ `ns_PosMaxFractionalDigits` specifies the maximum number of characters to the right of the decimal mark. If there are more characters than the maximum, the string is truncated.
- ◆ `ns_NegGroups` is similar to `ns_PosGroups`, except it specifies negative numbers.
- ◆ `ns_NegGroupSep` is similar to `ns_PosGroupSep` but for negative numbers.
- ◆ `ns_NegRadix` is similar to `ns_PosRadix` except it specifies negative numbers.
- ◆ `ns_NegFractionalGroups` is similar to `ns_PosFractionalGroups`, except it specifies negative numbers.
- ◆ `ns_NegFractionalGroupSep` is similar to `ns_PosFractionalGroupSep`, except it specifies negative numbers.
- ◆ `ns_NegFormat` similar to `ns_PosFormat`, except it specifies negative numbers.
- ◆ `ns_NegMinFractionalDigits` is similar to `ns_PosMinFractionalDigits`, except it specifies negative numbers.
- ◆ `ns_NegMaxFractionalDigits` is similar to `ns_PosMaxFractionalDigits`, except it specifies negative numbers.
- ◆ `ns_Zer` If the number being processed is 0, then this string pointer is used as is and is copied directly into the resulting buffer. If this field is NULL, then the number is formatted as if it were a positive number.

Typically, an application doesn't have to deal with the interpretation of a `NumericSpec` structure. The `Locale` structure contains three `NumericSpec` structures initialized with the correct information for the target culture, which can be passed to `intlFormatNumber()` to convert numbers into string form.

DateSpec Arrays

Dates also vary in format across different cultures. The `Locale` structure contains four `DateSpec` arrays which define the local formats for date and time representation.

The `DateSpec` arrays contain format commands similar to `printf()` format strings. The `"%"` commands are inserted in the formatting string to produce custom date output. An application can provide the `DateSpec` structure from the `Locale` structure to the `intlFormatDate()` function or it can create its own `DateSpec` structures for custom formatting.

For more information on `DateSpec` arrays, see the reference documentation of the `intlFormatDate()` function.

Using the Locale Structure

The main use of the International folio is to obtain the target language and country codes. The folio also provides functions to format dates, numbers, and currency for the target language and country. This section describes how to use these functions.

Determining the Current Language and Country

The `intlOpenLocale()` call determines the current user settings for language or country:

```
Item intlOpenLocale(const TagArg *tags)
```

This call accepts one argument, `tags`, a pointer to a tag argument list. This pointer is currently unused and must be set to `NULL`. `intlOpenLocale()` returns an `Item`.

You can then use the following call to obtain the `Locale` structure for the item:

```
Locale *intlLookupLocale(Item locItem)
```

This call accepts one argument, `locItem`, an item as returned from `intlOpenLocale()`. The language code in the `Locale` structure can then determine which set of messages and artwork to use.

For example, if the language code were `"de,"` your program could use a directory containing the German message text and artwork (`"de"` being the code for Deutsch). There would be a directory for every language your program supports.

Note: *An application must supply any text or artwork it displays. The application must supply text strings or artwork for each language it supports.*

When you finish using a locale, call the following function to terminate your use of it:

```
Err intlCloseLocale(Item locItem)
```

`intlCloseLocale()` accepts one argument, `locItem`, the item to be closed. It returns zero if the call was successful, otherwise, an error code is returned.

Working With International Character Strings

Comparing Character Strings

Use the `intlCompareStrings()` function to compare strings and sort text to be displayed. Each language sorts text in different ways. `intlCompareStrings()` adapts its behavior to the current language. By using this function, your title sorts text in the appropriate manner for the target culture:

```
int32 intlCompareStrings(Item locItem, const unichar *string1,  
    const char *string2)
```

This call accepts three arguments: `locItem` is a `Locale` item obtained from `intlOpenLocale()`, `string1` is a pointer to the first string to compare, and `string2` is a pointer to the second string to compare.

The comparison is performed according to the collation rules of `locItem`. `intlCompareStrings()` is similar to the standard `C strcmp()` function, except that it handles sorting variations of different languages.

`IntlCompareStrings()` returns -1 if `string1` is less than `string2`; 0 if `string1` is equal to `string2`; and 1 if `string1` is greater than `string2`. `INTL_ERR_BADITEM` is returned if `locItem` was not a valid `Item`.

Changing Letters In Character Strings

`intlConvertString()` strips diacritical marks and changes the letter case of words or characters. This function handles the language differences in rules for case conversion. It is similar to the standard `C toupper()` and `tolower()` functions.

```
int32 intlConvertString(Item locItem, const unichar *string, unichar  
    *result, uint32 resultSize, uint32 flags)
```

The call accepts five arguments: `locItem` is an item returned from `intlOpenLocale()`, `string` is the string to be changed, `result` is where the result of the conversion is placed, `resultSize` is the number of bytes available in `result`, and `flags` indicates the conversion to be performed. The string copied to `result` is guaranteed to be NULL-terminated.

If successful, `intlConvertString()` returns a positive number indicating the number of characters in `result`. Otherwise, it returns an error code.

Changing Character Sets

`intlTransliterateString()` converts a string from one character set to another character set.

The `intlTransliterateString()` function can be used to convert data files generated in ASCII to UniCode or to convert the output of the International folio into ASCII for use by other tools.

`intlTranliterateString()` is called as follows:

```
int32 intlTransliterateString(const void *string, CharacterSets
    stringSet, void *result, CharacterSets resultSet, uint32
    resultSize, uint8 unknownFiller)
```

The call accepts six arguments: `string` is the string to convert, `stringSet` is the character set of the string to convert, `result` is where to put the converted string, `resultSet` is the character set to use for the resulting string, `resultSize` is the number of bytes available in `result`, and `unknownFiller` is used as filler for characters that cannot be converted.

If successful, `intlTransliterateString()` returns a positive number indicating the number of characters in `result`. Otherwise, it returns an error code.

Providing C Functionality

`intlGetCharAttrs()` provides functionality similar to the standard C functions `isupper()`, `islower()`, etc. It is called as follows:

```
uint32 intlGetCharAttrs(Item locItem, unichar character)
```

The call accepts two arguments: `locItem`, an item as returned by `intlOpenLocale()`; and `character`, the character for which attributes should be returned.

If successful, `intlGetCharAttrs()` returns a bit mask which indicates the attributes of the character. `INTL_ERR_BADITEM` is returned if `locItem` is not a valid item.

Formatting Numbers or Currency

Numbers are represented in different ways in different countries, so a title should use the following call to format a number to be displayed in the correct manner:

```
int32 intlFormatNumber(Item locItem, const NumericSpec *spec, uint32
    whole, uint32 frac, bool negative, bool doFrac, unichar *result,
    uint32 resultSize)
```

The call accepts seven arguments: `locItem` is an item returned from `intlOpenLocale()`, `spec` is the format specification for the number (usually taken from the `Locale` structure), `whole` is the whole component of the number, `frac` is the decimal component of the number expressed in billionths (to the right of the decimal mark), `negative` is a Boolean that indicates whether the number is negative, `doFrac` is a flag indicating which portions of the number should be formatted (if `TRUE`, the entire number with a decimal mark is output, if `FALSE`, only the whole part of the number is output), `result` is where the formatted number is put, and `resultSize` is the number of bytes in `result`.

The number is formatted according to the rules specified in `locItem` and `spec`. The string copied to `result` is guaranteed to be NULL-terminated.

If successful, `intlFormatNumber()` returns a positive number indicating the number of characters in `result`. Otherwise, it returns an error code.

Formatting Dates

An application should use the following call to format a date to be displayed:

```
int32 intlFormatDate(Item locItem, DateSpec spec, const GregorianCalendar
    *date, unichar *result, uint32 resultSize)
```

The call accepts five arguments: `locItem` is an item as returned from `intlOpenLocale()`, `spec` is the format specification for the date (usually taken from the `Locale` structure), `date` is the date to format, `result` is where to put the formatted date, and `resultSize` is the number of bytes in `result`. The date is formatted according to the rules of `locItem` and `spec` arguments. The string copied into `result` is guaranteed to be NULL-terminated.

If successful, `intlFormatDate()` returns a positive number indicating the number of characters in `result`. Otherwise, an error code is returned.

The Compression Folio

This chapter describes the Compression folio that provides general purpose compression and decompression services.

This chapter contains the following topics:

Topic	Page Number
Introduction	227
How the Compression Folio Works	228
How to Use the Compression Folio	228
The Callback Function	230
Controlling Memory Allocations	230
Convenience Calls	231
Example: Using Compression	231

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Introduction

This section provides an overview of the Compression folio and describes how it works. See *3DO System Programmer's Reference* for complete descriptions of the calls mentioned in this section.

The Compression folio provides services that compress and decompress data in a non-lossy manner. The compression quality differs depending on the type of data. Compression ratios range typically from 50 percent up to 80 percent.

How the Compression Folio Works

The sections below describe how to compress and decompress data.

Compressing Data

To compress data, you must create a compression engine by calling `CreateCompressor()`.

You supply the function with a pointer to an output function.

Once you create the engine, call `FeedCompressor()` to have the engine compress the data you supply. `FeedCompressor()` then gives data to the compression engine, which in turn sends it to your output function, one word at a time. When the compression engine is finished compressing the data, call `DeleteCompressor()` to delete the engine and clean up.

Decompressing Data

Decompression works almost the same as compressing data. First, you create a decompression engine with `CreateDecompressor()`. Then provide a pointer to your output function.

Feed compressed data to the decompression engine with the `FeedDecompressorEngine()` call, which in turn sends it to the output call. Again when you are finished decompressing data, call `DeleteDecompressor()` to delete the engine and clean up.

How to Use the Compression Folio

To use the Compression folio to compress data, an application must do the following:

1. Open the Compression folio:

```
Err OpenCompressionFolio(void);
```

2. Create a compression engine:

```
Err CreateCompressor(Compressor **comp, CompFunc cf, const TagArg  
*tags);
```

When creating a compression engine, you must supply a pointer to a `Compressor` variable. This is where the handle to the compression engine will be stored. In addition, you must also provide a function pointer. This function is called automatically by the compression engine whenever a word

of compressed data is generated. The function is then responsible for taking the word of data and putting it somewhere useful for the application.

3. Feed data to the compression engine by calling:

```
Err FeedCompressor(Compressor *comp, void *data, uint 32
numDataWords);
```

As you feed data to the compressor, it compresses this data and calls the output function for every word of data it generates.

4. Delete the compression engine when all of the data is compressed.

```
Err DeleteCompressor(Compressor *comp);
```

Deleting the compression engine flushes any data left to output, and deallocates any resources allocated when the compression engine was created.

5. Close the Compression folio:

```
CloseCompressionFolio();
```

To use the Compression folio to decompress data, an application must do the following:

1. Open the Compression folio:

```
Err OpenCompressionFolio(void);
```

2. Create a decompression engine:

```
Err CreateDecompressor(Decompressor **decomp, CompFunc cf, const
TagArg *tags);
```

When creating a decompression engine, you must supply a pointer to a Decompressor variable. This is where the handle to the decompression engine is stored. In addition, you must also provide a function pointer. This function is called automatically by the decompression engine whenever a word of decompressed data is generated. The function is then responsible for taking the word of data and putting it somewhere useful for the application.

3. Feed data to the decompression engine by calling:

```
Err FeedDecompressor(Decompressor *decomp, void *data, uint 32
numDataWords);
```

As you feed data to the decompressor, it decompresses this data and calls the output function for every word of data it generates.

4. Delete the decompression engine when all of the data is decompressed.

```
Err DeleteDecompressor(Decompressor *decomp);
```

Deleting the decompression engine flushes any data left to output, and deallocates any resources allocated when the decompression engine was created.

5. Close the compression folio:

```
CloseCompressionFolio();
```

The Callback Function

When you create a compression or decompression engine, you must supply a function pointer. This function is called whenever either engine has data to output. The function then takes the data and does whatever the application wants with the data. For example, the callback function can deposit the data in a buffer, and once the engine is deleted, the buffer can be written out to a file.

The callback function has two parameters passed to it, and is defined as:

```
typedef void (* CompFunc)(void *userData, uint32 word)
```

The `userData` field is normally `NULL`, unless you supply the `COMP_TAG_USERDATA` tag when creating the engine. Whatever value is associated with the tag gets passed directly to the callback function. The purpose of the `userData` field is to pass things like a file pointer to the callback function. You would pass the file pointer as the data value associated with the `COMP_TAG_USERDATA` tag. The engines then pass the value to the callback function whenever it is called.

The `word` parameter to the callback is the word of data being generated by the engine. You must do something useful with this word. For example, you could copy it into a memory buffer, or write it to a file.

Controlling Memory Allocations

When you create a compression or decompression engine, the `CreateCompressor()` and `CreateDecompressor()` functions normally allocate some memory to hold the state of the engines. The compression engine needs around 28 KB of storage, while the decompressor needs around 5 KB.

If your title has sophisticated memory management needs, you can supply a memory buffer yourself to avoid having the folio allocate any memory. To do this, call the `GetCompressorWorkBufferSize()` routine to obtain the size of the buffer needed for the compression engine, or you can call `GetDecompressorWorkBufferSize()` to get the size of the buffer needed for the decompression engine. Once you have determined the size, then you can allocate a memory buffer, and pass a pointer to the buffer using the `COMP_TAG_WORKBUFFER` tag when you create the compressor or decompressor.

When you supply the work buffer, the Compression folio allocates no, or almost no, resources when a compression or decompression engine is created. This allows your title to control where memory resources come from.

Convenience Calls

Portfolio supplies two function calls that provide a simple interface to the lower-level functions of the Compression folio. The two functions implement the most common uses for the folio functions.

The `SimpleCompress()` function compresses data from one memory buffer into another memory buffer. You give it a source buffer pointer, the size of the source buffer, a destination buffer pointer, and the size of the destination buffer. `SimpleCompress()` then compresses the data and deposits the compressed result in the destination buffer. This function returns the number of words of data copied into the destination buffer, or a negative error if something went wrong, such as a lack of memory or a lack of space in the destination buffer.

The `SimpleDecompress()` function decompresses in much the same way as `SimpleCompress()` compresses data. This time, the source buffer contains the compressed data, and the destination buffer gets filled with decompressed data. This function returns the number of words of decompressed data that were put in the destination buffer, or a negative error code if something went wrong.

Example: Using Compression

Example 15-1 contains the sample program *compressexample.c* (located in the Examples folder), which shows how to use the Compression folio. This program loads itself into a memory buffer, compresses the data, decompresses it, and finally compares the original data with the decompressed data to make sure the compression process was successful.

Example 15-1 *Compressing and decompressing data (compression.c).*

```
/*
**
** Copyright (c) 1993-1996, an unpublished work by The 3DO Company.
** All rights reserved. This material contains confidential
** information that is the property of The 3DO Company. Any
** unauthorized duplication, disclosure or use is prohibited.
**
** @(#) compression.c 95/09/30 1.5
** Distributed with Portfolio V30.0
**
** */
```

```
#include <:kernel:types.h>
#include <:kernel:mem.h>
#include <:kernel:time.h>
#include <:file:fileio.h>
#include <:misc:compression.h>
#include <stdio.h>
#include <stdlib.h>

/*****

int main(int32 argc, char **argv)
{
    RawFile    *file;
    FileInfo    info;
    Err         err;
    bool        same;
    uint32      i;
    int32       fileSize;
    uint32      *originalData;
    uint32      *compressedData;
    uint32      *finalData;
    int32       numFinalWords;
    int32       numCompWords;
    TimerTicks  compStartTime, compEndTime, compTotalTime;
    TimerTicks  decompStartTime, decompEndTime, decompTotalTime;
    TimeVal     compTV, decompTV;

    TOUCH(argc);

    err = OpenCompressionFolio();
    if (err >= 0)
    {
        err = OpenRawFile(&file,argv[0],FILEOPEN_READ);
        if (err >= 0)
        {
            GetRawFileInfo(file,&info,sizeof(info));
            fileSize      = info.fi_ByteCount & 0xffffffff;
            originalData   = (uint32 *)malloc(fileSize);
            compressedData = (uint32 *)malloc(fileSize);
            finalData      = (uint32 *)malloc(fileSize);

            if (originalData && compressedData && finalData)
            {
```

```

    if (ReadRawFile(file, originalData, fileSize) == fileSize)
    {
        SampleSystemTimeTT(&compStartTime);

        err = SimpleCompress(originalData, fileSize /
sizeof(uint32),
compressedData, fileSize / sizeof(uint32));

        SampleSystemTimeTT(&compEndTime);

        if (err >= 0)
        {
            numCompWords = err;

            SampleSystemTimeTT(&decompStartTime);

            err = SimpleDecompress(compressedData, numCompWords,
                                finalData,
                                fileSize / sizeof(uint32));

            SampleSystemTimeTT(&decompEndTime);

            if (err >= 0)
            {
                SubTimerTicks(&compStartTime, &compEndTime,
&compTotalTime);
                SubTimerTicks(&decompStartTime,
&decompEndTime, &decompTotalTime);
                ConvertTimerTicksToTimeVal
(&compTotalTime, &compTV);
                ConvertTimerTicksToTimeVal
(&decompTotalTime, &decompTV);
                numFinalWords = err;

                printf("Original data size      : %d words\n",
fileSize / sizeof(uint32));
                printf("Compressed data size   : %d words\n",
numCompWords);
                printf("Uncompressed data size: %d words\n",
numFinalWords);
                printf("Compression Time       : %d.%06d\n",
compTV.tv_Seconds, compTV.tv_Microseconds);
                printf("Decompression Time      : %d.%06d\n",
decompTV.tv_Seconds, decompTV.tv_Microseconds);

                same = TRUE;
                for (i = 0; i < fileSize / sizeof(uint32); i++)

```

```
        {
            if (originalData[i] != finalData[i])
            {
                same = FALSE;
                break;
            }
        }

        if (same)
        {
            printf("Uncompressed data matched original\n");
        }
        else
        {
            printf("Uncompressed data differed with
original!\n");
            for (i = 0; i < 10; i++)
            {
                printf("orig %08x, final %08x, comp
                %08x\n",
                        originalData[i],
                        finalData[i],
                        compressedData[i]);
            }
        }
    }
    else
    {
        printf("SimpleDecompress() failed: ");
        PrintfSysErr(err);
    }
}
else
{
    printf("SimpleCompress() failed: ");
    PrintfSysErr(err);
}
}
else
{
    printf("Could not read whole file\n");
}
}
else
{
    printf("Could not allocate memory buffers\n");
}
```

```
        free(originalData);
        free(compressedData);
        free(finalData);

        CloseRawFile(file);
    }
    else
    {
        printf("Could not open '%s' as an input file: ",argv[0]);
        PrintfSysErr(err);
    }
    CloseCompressionFolio();
}
else
{
    printf("OpenCompressionFolio() failed: ");
    PrintfSysErr(err);
}

return (0);
}
```

The IFF Folio

This chapter describes the IFF folio, which contains functions to read and write data stored in the IFF format.

This chapter contains the following topics:

Topic	Page Number
Overview of the IFF File Format and Folio	237
Description of the IFF File Format	243
Using the IFF Folio - a Tutorial	250
Description of IFF Folio Functions	255

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Overview of the IFF File Format and Folio

The EA IFF 85 Standard

IFF refers to the *EA IFF 85 Standard for Interchange Format Files*, originally developed by Electronic Arts. The IFF standard is now used by 3DO and a number of other companies, primarily to store multimedia data.

IFF is a standard file format and set of conventions for storing composite data -- i.e., data that consists of multiple related elements. Some examples might be

- ◆ An audio file that contains a sound sample and related control data.

- ◆ A raster image file that contains header information along with a color map and the raster data.
- ◆ An animation file that contains picture frames, music, and control data.

The purpose of the IFF standard is to allow interchange of files between different applications and to simplify the process of reading and parsing composite files.

The IFF File Format

An IFF file consists of a container "chunk," which contains one or more enclosed local "chunks." The container chunk corresponds to the file. The local chunks are blocks of data within the file.

A container chunk consists of the following:

- ◆ A 4-byte ID field indicating the format of the container chunk. "FORM" is the most common format. There are also the "CAT" and "LIST" formats, which will be discussed later.
- ◆ A 4-byte size field specifying the total number of bytes in the data portion of the container chunk.
- ◆ A 4-byte type field specifying the type of data in the container chunk.
For example, the type field for one kind of audio data file is "AIFC". This kind of file contains audio samples, usually in a compressed format.
- ◆ The container chunk's data, which consists of one or more local chunks.

Each local chunk consists of the following:

- ◆ A 4-byte ID field indicating the kind of data in the local chunk.
In an AIFC file, for example, the chunk containing a sound sample has an ID field of "SSND".
- ◆ A 4-byte length field indicating the number of bytes of data in the local chunk.
- ◆ The local chunk's data, which in the case of a SSND chunk, contains a sound sample.

The illustration in Figure 16-1 shows an example of an AIFC FORM file.

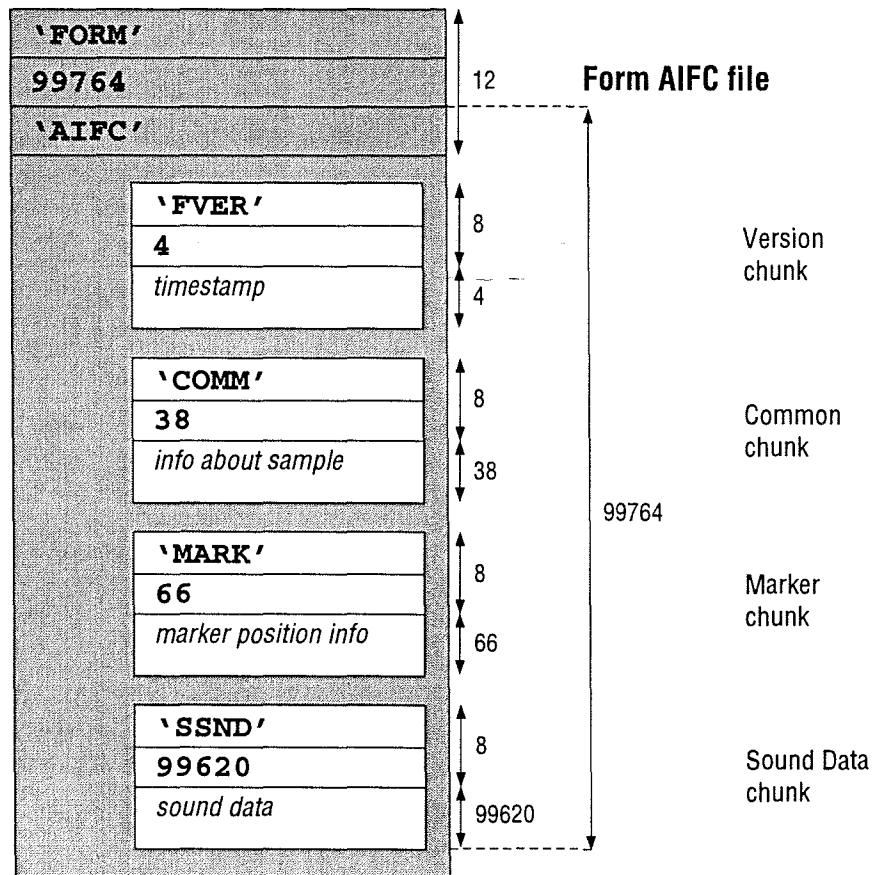


Figure 16-1 Layout of a sample AIFC FORM

A container chunk can contain embedded (i.e., nested) container chunks. A FORM can contain other FORMs, which in turn contain FORMs, and so on.

See "Description of the IFF File Format" on page 243 for more detail on the IFF file format.

Goals of the IFF Format

The IFF format was developed with the following goals:

- ◆ Provide a standard methodology for storing structured, composite data of the kind commonly used for multimedia.
- ◆ Make it easier to parse composite data.
- ◆ Make it easier to interchange multimedia files between different applications.

- ◆ Allow a composite file to be self sufficient and self-defining -- i.e., to contain all the information that different applications would need to process it.
The goal is to not require an application to know ahead of time the number of channels for an audio sample, for example. This information is contained in the file.
- ◆ Allow an application to find chunks of data that are relevant to its specific function and ignore the rest without having to scan all the data in the file. The size field in each chunk makes this easy.
- ◆ Allow the development of standard IFF chunk types, such as the AIFC FORM, used to store audio samples, and the TXTR FORM, used to store textures (i.e., pictures).

The Purpose and Capabilities of the IFF Folio

Purpose of IFF Folio

Processing an IFF file involves opening the container chunk and then identifying and handling each chunk inside it. This involves reading each chunk's ID and length fields and then processing the chunk's data or skipping over it to get to the next chunk.

You can write your own logic to parse through IFF files, or you can use the IFF folio provided by 3DO. The IFF folio is essentially an IFF parsing machine. It is a set of functions that "understand" the structure of IFF files and provide a basic set of operations that enable your application to process an IFF file efficiently.

General Capabilities of the Functions in IFF Folio

The IFF folio provides functions that let you do the following:

- ◆ Tell the IFF parser what action to take when it encounters a certain type of chunk (e.g., COMM, MARK, SSND, etc.):
 - ◆ Stop at the beginning of the chunk and return control to your application to let your application process the data in the chunk -- e.g., play the sound sample in a SSND chunk.
 - ◆ Save the chunk in a linked list for use when processing chunks later on in the stream. You might do this for a chunk, such as a COMM chunk, that contains control information used in processing other chunks in the FORM.
 - ◆ Invoke a callback function to do some application-specific processing before continuing to parse.
- ◆ Establish and stack context information.

When you are processing a chunk, you may want to use information from the chunk itself, or information from its parent container chunk, or even information from the parent container chunk's parent container chunk.

The IFF folio lets you organize these contexts in a simple stack. When the IFF parser enters a chunk, it creates a context node for that chunk and pushes that context node onto the context stack. When the IFF parser exits a chunk, it pops the chunk's context node from the stack.

The IFF folio also provides functions that let you attach context information to a node and then find that information easily later on.

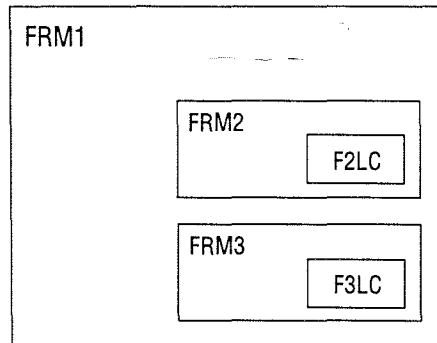


Figure 16-2 A FORM with 2 embedded FORMs containing local chunks

Look at the example FORM shown in Figure 16-2 above. This FORM, FRM1, contains 2 embedded FORMs, FRM2 and FRM3, each of which contains a local chunk.

- ◆ Before the first chunk is processed, the IFF parser creates an initial context node, known as the “default context node”, which is always at the bottom of the stack. The default context node is embedded in the parser structure that controls the parsing operation and is not popped until the parser structure is deleted.
- ◆ When you begin processing the FORM FRM1, the IFF parser creates a context node for FRM1 and pushes the node onto the context stack on top of the default context node.
- ◆ When you enter the embedded FORM FRM2, the IFF parser creates a context node for FRM2 and pushes the node onto the context stack on top of FRM1's context node.
- ◆ When you enter FRM2's local chunk, F2LC, the IFF parser creates a context node for F2LC and pushes that node onto the context stack on top of FRM2's context node. See Figure 16-3 below.

While reading and processing the data in F2LC, you can use IFF functions to locate relevant context information in the FRM2, FRM1, and default contexts.

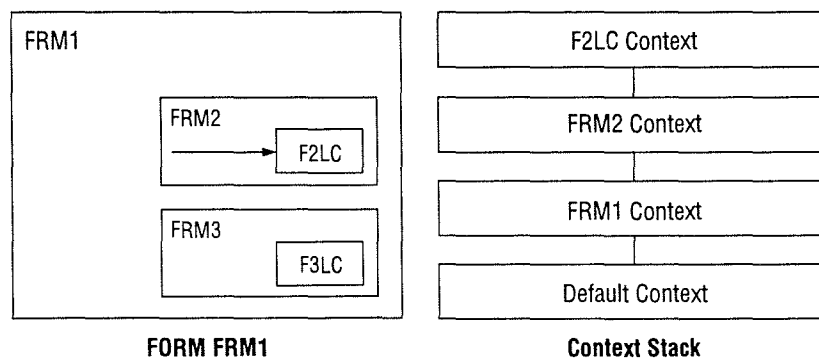


Figure 16-3 Context stack when local chunk F2LC is being read

- ◆ When you exit local chunk F2LC, the parser pops F2LC's context node from the stack and gets rid of it.
- ◆ When you exit the embedded FORM FRM2, the parser pops FRM2's context node from the stack and gets rid of it.
- ◆ When you enter the next embedded FORM, FRM3, the IFF parser creates a context node for FRM3 and pushes the node onto the context stack on top of FRM1's context node.
- ◆ When you enter FRM3's local chunk, F3LC, the IFF parser creates a context node for F3LC and pushes that node onto the context stack on top of FRM3's context node. See Figure 16-4 below.

While reading and processing the data in F3LC, you can use IFF functions to locate context information in the FRM3, FRM1, and default contexts.

- ◆ And so on.

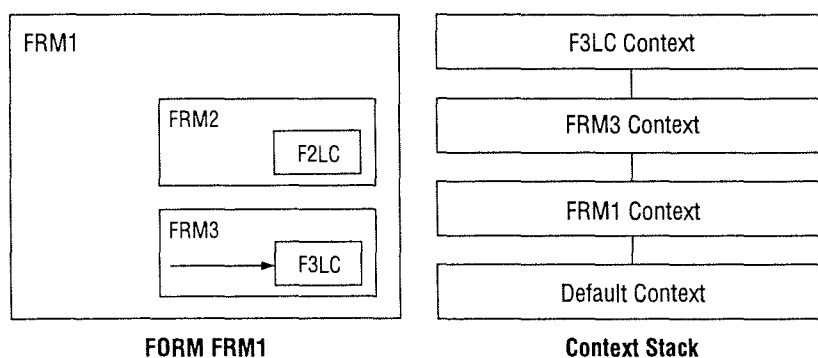


Figure 16-4 Context stack when local chunk F3LC is being read

- ◆ The IFF folio also lets you read a chunk into storage.
- ◆ And the IFF folio lets you write a chunk from storage to a file system.

See “Using the IFF Folio - a Tutorial” on page 250 for a tutorial on using the IFF folio and “Description of IFF Folio Functions” on page 255 for descriptions of the individual functions.

How IFF Formats and Folio are Used in 3DO M2 System

The 3DO M2 system uses the IFF format for many different kinds of data. Some examples are

- ◆ Icons
- ◆ Saved game data
- ◆ Sound samples
- ◆ Textures
- ◆ Fonts

In most cases, your application accesses the data through higher-level API functions that insulate your application from the details of IFF file format. These API functions call the IFF folio functions for you.

Some examples are

- ◆ Icon folio functions, such as `SaveIcon()` and `LoadIcon()`, which write icons out to a file system and read icons into memory.
- ◆ SaveGame folio functions, such as `SaveGameData()` and `LoadGameData()`, which write saved game data out to a filesystem and read saved game data into memory.
- ◆ `LoadSample()`, which loads a sound sample (AIFF or AIFC FORM) from an AIFF or AIFC file.
- ◆ `Spr_LoadTexture()`, which loads textures (TXTR FORMs) from a UTF file.
- ◆ `OpenFont()`, which loads a 3DO font file into memory.

Description of the IFF File Format

This section describes the IFF file format in more detail.

Every IFF file consists of a container chunk that contains other chunks. The internal chunks can be local chunks or embedded container chunks.

Standard Chunk Format

All IFF chunks have the following format:

Example 16-1 *Format of IFF chunk*

```
typedef struct {  
    PackedID ckID ;           /* chunk ID 4 bytes*/  
    uint32 ckDataSize ;       /* bytes of data in ckData */  
    char ckData[];           /* data (of any type) */  
} Chunk;
```

- ◆ The chunk ID (ckID) must be 4 ASCII characters in the range hex 20 (space) through hex 7E (tilde).

No leading spaces are permitted.

IDs are compared using a quick, case-sensitive 32-bit equality test.

The ID field of a container chunk is one of 4 special chunk IDs ("FORM", "CAT ", "LIST", "PROP").

The ID field of a local chunk can be any 4-character name.

- ◆ The length of the data contained in a chunk's data portion (ckData) is described by is a 32-bit unsigned integer (ckDataSize).
- ◆ The data (ckData) can be any kind of data.

The first 4 bytes of a container chunk's (e.g., FORM's) data portion contain a 4-character field indicating the type of data in the chunk (e.g., "AIFC", "TXTR").

The data bytes after the type field are the local and embedded chunks contained in the container chunk. See "Layout of a sample TXTR FORM" on page 245.

Note that for a container chunk, the type of data it contains is indicated by the type field in the first 4 bytes of its data portion (e.g., "AIFC", "TXTR"), not by the chunk ID, which must be "FORM", "CAT ", "LIST", or "PROP".

For a local chunk, the type of data is indicated by the chunk ID (e.g., "COMM", "SSND").

- ◆ Every odd-length chunk must be followed by a pad byte of zeroes so that the total length of chunk plus pad byte is even. The pad byte is not included in the chunk's data size. See "Layout of a sample TXTR FORM" on page 245.
- ◆ Every 16-bit and 32-bit number is stored in "big-endian" byte order — highest byte first.

An Intel CPU must reverse the 2- or 4-byte sequence of each number. This applies to chunk dataSize fields and to numbers inside chunk data. It does not apply to character strings and byte data because you can't reverse a 1-byte sequence.

- ◆ Every 16-bit and 32-bit number is stored on an even address.

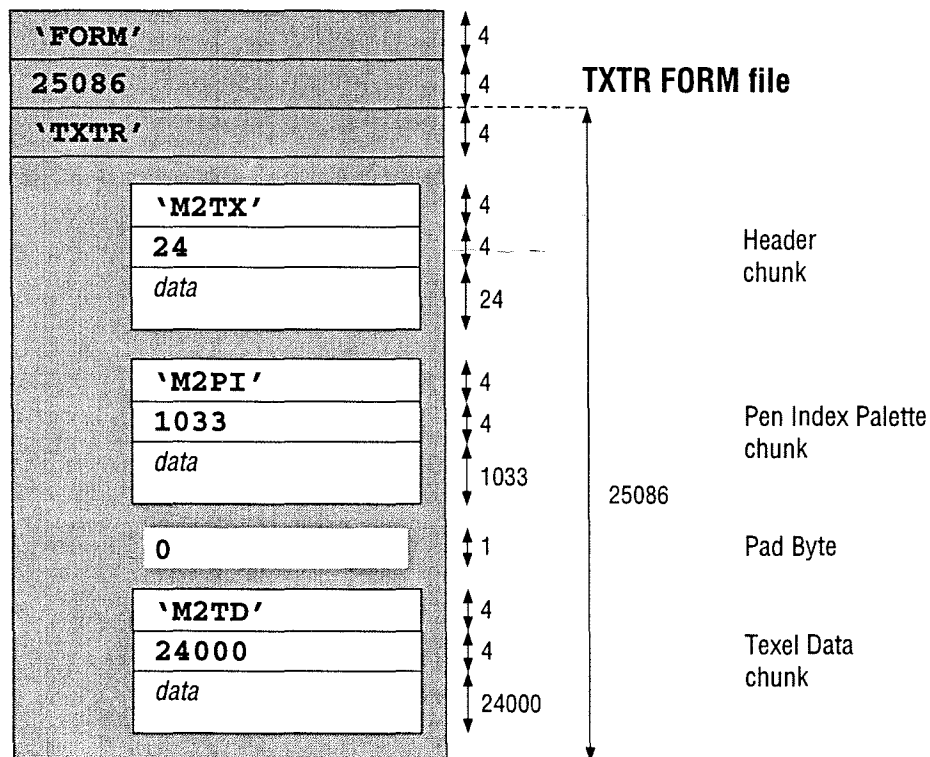


Figure 16-5 Layout of a sample TXTR FORM

3DO Extension to Standard Chunk Format

3DO provides an extension to the standard format to allow chunks whose size is larger than can be contained in a 32-bit field.

If the size field in a chunk is equal to the value `IFF_SIZE_64BIT` (a negative number), then the size of the chunk's data is contained in the next 64 bits. This is then followed by the chunk's data.

This means that, in a container chunk, the type field (e.g., "AIFC") is 16 bytes from the beginning of the container rather than the standard 8 bytes.

See Figure 16-6 below.

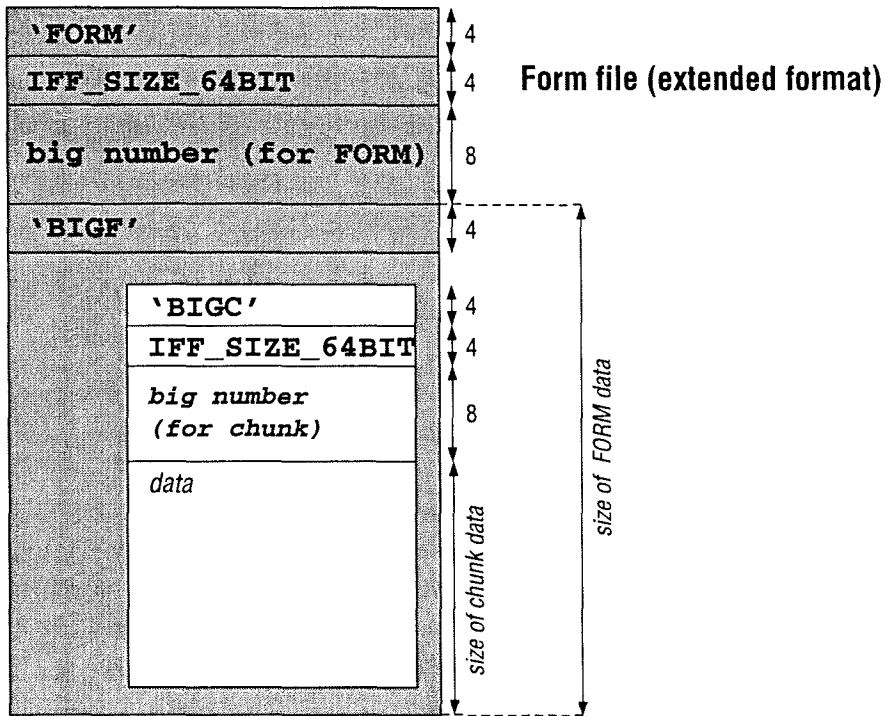


Figure 16-6 3DO extended format

Note that the 3DO extended format is not currently used very often. Furthermore, it will usually be more or less transparent to your application because IFF functions handle it automatically. The structures that represent chunks in memory always use a 64-bit field to contain a chunk's size regardless of whether the chunk was in standard or extended format in the file system.

Container Chunks

There are four kinds of container chunks: FORM, CAT, LIST, and PROP. Every IFF file is a FORM, CAT, or LIST chunk. PROP chunks can only be used as embedded chunks at the beginning of LIST chunks.

FORM Chunks

A FORM chunk begins with the special 4-byte chunk ID "FORM". This is followed by the 4-byte data size field, which is followed by the data itself. The first 4 bytes of data are a 4-byte character field (e.g., "AIFC") indicating the type of data that is contained in the FORM. This is followed by the chunks contained in the FORM.

FORM chunks are the most commonly used container chunks. Most IFF files consist of a single FORM chunk containing a set of local chunks.

However, a FORM chunk can contain embedded FORM chunks. The level of nesting can be infinite, theoretically.

CAT Chunks

The purpose of a CAT chunk is to let you concatenate multiple FORM or LIST chunks, one after another, within a single container chunk. In effect, a CAT chunk acts like a library of FORM or LIST chunks. For example, you might use a CAT chunk to contain a library of textures, each texture being a TXTR FORM chunk.

A CAT chunk begins with the special 4-byte chunk ID "CAT ". This is followed by the 4-byte data size field, which is followed by the data itself. The first 4 bytes of data are a 4-byte character string (e.g., "TXTR") indicating the type of FORM or LIST chunks contained in the CAT. This is followed by the FORM or LIST chunks themselves.

An example of a CAT chunk used as a TXTR texture library is shown below in outline form.

Example 16-2 *Outline of a CAT chunk containing FORM chunks*

CAT TXTR

FORM TXTR
 M2TX
 M2PI
 M2TD

FORM TXTR
 M2TX
 M2PI
 M2TD

LIST and PROP Chunks

A LIST chunk normally contains one or more PROP chunks followed by FORM chunks (or, rarely, LIST or CAT chunks).

A PROP chunk contains local chunks with global default data that can be used by the ensuing FORM chunks. Each FORM chunk can omit local chunks that are provided by the PROP chunks, or it can override them with local chunks.

PROP chunks can only appear in LIST chunks and must always be at the beginning of a LIST chunk. A PROP chunk begins with the special 4-byte chunk ID "PROP". This is followed by the 4-byte data size field, which is followed by the data itself.

The first 4 bytes of data are a 4-byte character field (e.g., "TXTR") indicating the type of FORM chunks for which the PROP chunk contains default data. This type indicator is followed by the PROP chunk's local chunks, which can be used or overridden by the ensuing FORM chunks.

An example of a LIST chunk used as a TXTR texture library, with a PROP chunk containing a global header (M2TX) and pen index palette (M2PI), is shown below in outline form.

Example 16-3 *Outline of a LIST chunk containing PROP and FORM chunks*

```
LIST TXTR

    PROP TXTR
        M2TX
        M2PI

    FORM TXTR
        M2TD

    FORM TXTR
        M2PI
        M2TD
```

Note that the second FORM chunk contains a local Pen Index Palette (M2PI) chunk, which overrides the global Pen Index Palette chunk contained in the PROP chunk.

Global and Local Chunk IDs

The special chunk IDs (i.e., FORM, CAT, LIST, PROP) and the container type IDs (e.g., AIFC, TXTR) of container chunks are global. They mean the same thing wherever they appear in all files that follow the IFF-85 convention. An AIFC FORM is not a sound sample in one place and a raster image in another.

The IDs of local chunks, however, are normally local to the containing chunk. A SSND chunk contains sound data in an AIFC FORM, but could contain a completely different kind of data in a FOOB FORM.

However, 3DO has established a convention that can be used to define globally-scoped IDs for local chunks. The convention is that if a local chunk's ID begins with the character "!", its meaning is globally scoped. That chunk ID means the same thing, and a chunk with that ID has the same format in every container chunk in which it might appear.

This makes it possible to have standard chunks that serve the same purpose in every IFF file. At this time, the following global chunk IDs have been defined:

◆ **!Mod** = Modification Date chunk

Tells when the containing FORM was last written. The data bytes are a character string like "1995/02/01 15:05:14 PST".

Each field is fixed length. Some fields are optional. The format is:

yyyy/mm/dd[hh:nn[:ss][zzz]]

where yyyy is the year, mm is the month (1-12), dd is the day (1-31), hh is the hour (1-24), nn is the minute (1-60), ss is the seconds (1-60), and zzz is the time zone (e.g., UTC).

(Square brackets mean "optional". The slashes, spaces, and colons are literal.)

◆ **!(c)** = Copyright Notice chunk

Contains a copyright notice for the containing FORM.

The notice is a character string containing the desired copyright notice -- e.g., "Copyright (c) 1995 The 3DO Company".

◆ **!Rem** = Comment chunk

Provides a comment about the containing FORM.

This chunk contains a string, which may contain newline characters, such as ASCII Control-J characters, to separate lines.

◆ **!Ver** = Contents Version chunk

Contains a Release Version Number for the FORM's contents. This number is normally related to the modification date of the FORM's contents. Note that this is not the same as a Format Version number.

A Release Version Number is a character string containing a period-separated list of digits ordered from most to least significant. For example, "2.1.5" is equivalent to "2.1.05", which is less than "2.1.50". Each digit is non-negative and expressed in base 10.

◆ **!App** = Creator Application Name(s) chunk

Contains the name(s) of the application(s) used to create and edit the FORM's contents. This chunk contains a semicolon-separated list of application names, where each application name is a character string that can optionally be followed by a space and a version number.

The version number is used when providing customer technical support for that application; for example, "3DO Animator;PostPro 2.0".

- ◆ **!URL** = Uniform Resource Locator chunk
Contains a WWW URL string.

Using the IFF Folio - a Tutorial

This tutorial walks through a simple example. Your application will read in and write out a FORM chunk containing several local chunks. The goal is to show you the kinds of functions provided by the IFF folio, what they are used for, and the order in which they are generally used.

This tutorial will discuss each function in only enough detail to let you understand the general flow. Some of the IFF folio functions will not be used in the example. For more detail on the IFF folio functions, see "Description of IFF Folio Functions" on page 255.

Reading and Processing a FORM

The example FORM is of type VID1 (an imaginary, non-standard type) and contains 5 local chunks: 2 COL1 chunks, a COL2 chunk, a NOGO chunk, and a BODY chunk. Your application will read and parse this FORM chunk, handling the local chunks as follows:

- ◆ Save the contents of each COL1 chunk in a buffer that can easily be found later when processing the BODY chunk. The COL1 chunks contain information needed to process the BODY chunk.
- ◆ Save the contents of the COL2 chunk in a buffer that can easily be found later when processing the BODY chunk. The COL2 chunk also contains information that will be needed to process the BODY chunk.
- ◆ Read the first 40 data bytes of the NOGO chunk and then, depending on the contents, either
 - ◆ Stop reading the VID1 file, or
 - ◆ Store those 40 data bytes in a structure that can easily be found later when processing the BODY chunk. Then skip the rest of the NOGO chunk.
- ◆ Play the BODY chunk on a video device, using information from previous chunks to modify aspects of the display.

An outline of the VID1 FORM is shown in Example 16-4 below:

Example 16-4 *Outline of the VID1 FORM used for the tutorial*

```
FORM VID1
    COL1
    COL1
    COL2
    NOGO
    BODY
```

Overview

The basic steps involved in reading and processing a FORM file, using the IFF folio, are as follows:

1. Your application calls the IFF folio function `CreateIFFParser()` to open the file and create the parser structure that will control the operation.
2. Your application calls various IFF folio functions to tell the parser what to do with each kind of chunk in the FORM.
3. Your application calls the function `ParseIFF()` to begin the parsing operation. `ParseIFF()` scans the file, encountering each chunk and handling the chunk as you have directed.

If you did not specify any handling instructions for the chunks in the form, `ParseIFF()` would just scan through the FORM and exit.

4. Your application calls `DeleteIFFParser()` to delete the parser structure, close the file, and release resources.

Detail

Here is the sequence in more detail:

1. Your application calls the function `CreateIFFParser()` to allocate and initialize an `IFFParser` structure that will control this IFF parsing operation. The pointer to this structure is known as the parser handle for this parsing operation.

You pass as arguments a pointer to an address where `CreateIFFParser()` will store the parser handle, a boolean value indicating read mode, and a tag argument specifying the pathname of the file to read.

2. Your application calls the function `RegisterCollectionChunks()` to flag COL1 and COL2 as "collection" chunks to be "collected" -- i.e., saved in

linked lists in storage. There will be a linked list of COL1 chunks and a linked list of COL2 chunks.

3. Your application calls the function `InstallEntryHandler()` to point to a function in your application that you want to be called when the NOGO chunk is encountered.
4. Your application calls the function `RegisterStopChunks()` to flag BODY as a "stop" chunk, which means that, when the parser encounters a chunk with this ID, the parser should stop and return control to your application. Your application will then process the chunk.
5. Your application calls the function `ParseIFF()` to begin the parsing operation.

You pass as arguments the parser handle and a value telling `ParseIFF()` to parse in normal mode (`IFF_PARSE_SCAN`).

In normal mode, `ParseIFF()` will scan through the file, only returning to your application if a user handler returns an error code or if `ParseIFF()` encounters a stop chunk, an error, or the end of the file.

6. `ParseIFF()` encounters the VID1 FORM's header and pushes a FORM VID1 context node onto the context stack.
7. `ParseIFF()` encounters the first COL1 chunk's header, reads the chunk's data portion into a buffer, and creates a linked list whose first node (a `CollectionChunk` structure) points to the data buffer.
This linked list is anchored to the default context node, which is contained in the `IFFParser` structure.
8. `ParseIFF()` encounters the second COL1 chunk's header, reads its data into a buffer, and creates another `CollectionChunk` structure in the linked list of `CollectionChunk` structures for COL1 chunks. This new node, which is inserted at the head of the linked list, points to the second COL1 chunk's data.
9. `ParseIFF()` encounters the COL2 chunk's header, reads the chunk's data portion into a buffer, and creates a separate linked list anchored to the `IFFParser` structure. This list's first and only `CollectionChunk` structure points to the COL2 data buffer.
10. `ParseIFF()` encounters the NOGO chunk's header, pushes a context node for the NOGO chunk onto the context stack, and calls the entry handler specified by your application.
11. Your entry handler calls the function `ReadChunk()` to read the first 40 data bytes of the NOGO chunk into a buffer.

12. Your entry handler analyzes the 40 bytes and decides to continue. Then, your entry handler calls the function `AllocContextInfo()` to allocate a `ContextInfo` structure for the 40 bytes of info.
You pass as arguments the parser handle, the type of FORM chunk being processed (VID1), the ID of the current chunk (NOGO), an identifier of your choice indicating the type of data (let's use `GO_OR_NOGO`), and the number of bytes to allocate for the user buffer (40).
13. `AllocContextInfo()` allocates and initializes a `ContextInfo` structure and a data buffer. The `ContextInfo` structure points to the data buffer.
14. Your entry handler copies the 40 bytes of data into the data buffer pointed to by the `ContextInfo` structure.
Then your entry handler calls the function `StoreContextInfo()` to attach the `ContextInfo` structure to VID1's context node.
You pass as arguments the parser handle, a pointer to the `ContextInfo` structure, and a flag (`IFF_CIL_PROP`) that tells `StoreContextInfo()` to attach the `ContextInfo` structure to the context node of the top LIST or FORM chunk in the stack (VID1).
15. Your entry handler returns to `ParseIFF()`, returning a value (`IFF_CB_CONTINUE`) that tells `ParseIFF()` to continue parsing.
16. `ParseIFF()` encounters the BODY chunk's header, pushes a context node for the BODY chunk onto the context stack, and then returns to your application because BODY is a stop chunk.
17. Your application calls the function `ReadChunk()` to read the entire data portion of the BODY chunk into storage.
18. Your application then plays the data on the video device, while modifying the display using information from the two COL1 collection chunks, the COL2 collection chunk, and the `ContextInfo` structure created from the NOGO chunk.
 - ◆ To find the chain of COL1 collection chunks, your application calls the function `FindCollection()`, passing as arguments the parser handle, the FORM type ("VID1"), and the collection chunk ID ("COL1").
`FindCollection()`, returns a pointer to the first node in the linked list of `CollectionChunk` structures for COL1 chunks. Each `CollectionChunk` structure points to the data for a COL1 chunk. The first node is for the most recently encountered COL1 chunk.
 - ◆ To find the COL2 collection chunks, your application calls the function `FindCollection()`, passing as arguments the parser handle, the FORM type ("VID1"), and the collection chunk ID ("COL2").
`FindCollection()` returns a pointer to the linked `CollectionChunk` structure that points to the COL2 chunk.

- ◆ To find the `ContextInfo` structure created from information in the NOGO chunk, your application calls the function `FindContextInfo()`, passing as arguments the parser handle, the FORM type ("VID1"), the chunk ID ("NOGO"), and the identifier ("GO_OR_NOGO") that were specified in the `AllocContextInfo()` call.

`FindContextInfo()` returns a pointer to the `ContextInfo` structure.

19. Having completed its processing, your application calls the function `DeleteIFFParser()`, passing the parser handle.

The file is closed, and all resources are released.

Writing out a FORM

In this section, your application will write out a FORM chunk like the one shown in Example 16-4. To simplify this example, it will be assumed that all the data has been gathered and stored in buffers in memory.

Overview

The basic steps involved in writing out a FORM file, using the IFF folio, are as follows:

1. Your application calls the IFF folio function `CreateIFFParser()` to open the file and create the parser structure that will control the write operation.
2. For each chunk to be written out, your application calls the functions `PushChunk()`, `WriteChunk()`, and `PopChunk()` to
 - ◆ Push a context node for the chunk onto the context stack.
 - ◆ Write the chunk to the file.
 - ◆ Pop the chunk's context node off the context stack.
3. Your application calls the function `DeleteIFFParser()` to delete the parser structure, close the file, and release resources.

Detail

Here is the sequence in more detail:

1. Your application calls the function `CreateIFFParser()` to allocate and initialize an `IFFParser` structure that will control this IFF write operation.
You pass as parameters a pointer to a location where `CreateIFFParser()` will store the returned parser handle, a flag indicating write mode, and a tag argument specifying the pathname of the file to write to.
2. Your application calls the function `PushChunk()`, to create a context node for the VID1 FORM and push the node onto the context stack.

You pass as arguments the parser handle, the type of FORM being written ("VID1"), the chunk ID ("FORM"), and total number of data bytes that will be written for the FORM.

Normally, you just specify a size value of `IFF_SIZE_UNKNOWN_32`, which tells the parser to count the total number of data bytes written and then go back and write that value into the size field of the FORM chunk.

(Note that you would specify `IFF_SIZE_UNKNOWN_64` if you expected the FORM to contain more bytes than can be described by a 32-bit size field.)

`PushChunk()` creates the context node, pushes it onto the context stack, and writes the VID1 FORM's header to the file.

3. Your application calls the function `PushChunk()`, to create a context node for the first local chunk (COL1) and push the node onto the context stack.

You pass as arguments the parser handle, a null type field, the chunk ID for the local chunk being written (COL1), and the number of data bytes to be written for the chunk (or `IFF_SIZE_UNKNOWN_32` or `IFF_SIZE_UNKNOWN_64`).

`PushChunk()` creates the context node, pushes it onto the context stack, and writes the COL1 chunk's header to the file.

4. Your application calls the function `WriteChunk()` to write the COL1 chunk's data to the output file.

You pass as arguments the parser handle, a pointer to the buffer containing the data being written, and the number of bytes to write.

`WriteChunk()` writes the specified amount of data from the buffer to the file. If there is more than one buffer of data to write, your application calls `WriteChunk()` multiple times.

5. Your application calls the function `PopChunk()` to pop the context node for the COL1 chunk off the context stack.
6. Your application repeats this sequence of `PushChunk()`, `WriteChunk()`, `PopChunk()` for the rest of the chunks being written.
7. Your application calls the function `DeleteIFFParser()`, passing the parser handle.

Any data left to be written is written out, the file is closed, and all resources are released.

Description of IFF Folio Functions

This section provides a description of the IFF folio functions. For each function, this section provides a function prototype, a brief description of what the function does, and any interesting points about the function.

No attempt is made to describe all the arguments, tags, or details involved in using the function. For detailed reference information about using each function, see the *3DO M2 Portfolio Programmer's Reference*.

Setting Up, Starting, and Terminating a Parse Operation

Creating a Parser Structure

To create and initialize a new parser structure, call the `CreateIFFParser()` function.

```
Err CreateIFFParser(IFFParser **iff, bool writeMode,
                   const TagArg tags[])
```

You specify as arguments

- ◆ A pointer to a location in which `CreateIFFParser()` will store the parser handle. The parser handle is a pointer to the parser structure.
- ◆ A TRUE or FALSE value indicating whether the file will be written to.
- ◆ An pointer to an array of tag arguments that tell the parser where the IFF stream will be read from. You can read an IFF stream from a file or from another location in memory.
 - ◆ To read the stream from a file, specify the tag `IFF_TAG_FILE (const char *)`, with the pathname of the file to read.
 - ◆ To read the stream from memory, specify the tag `IFF_TAG_IOFUNCS (IFFIOFuncs *)`, with a pointer to a structure that contains pointers to custom callback functions provided by you. The parser calls these functions to do the "I/O" involved in reading the IFF stream from memory.

Starting or Continuing a Parse Operation

To start or continue a parsing operation, call the `ParseIFF()` function.

```
Err ParseIFF(IFFParser *iff, ParseIFFModes control)
```

You specify as arguments

- ◆ A pointer to the parser handle created by `CreateIFFParser()`.
- ◆ The parsing mode. Normally, you specify the value `IFF_PARSE_SCAN`, in which case the parser will only return control to your application when
 - ◆ It encounters an error.
 - ◆ It encounters a stop chunk.
 - ◆ It encounters the end of the logical file.
 - ◆ A user entry or exit handler returns a value other than 1.

After your application processes a stop chunk, your application continues the parsing operation by calling `ParseIFF()` again.

Deleting a Parser Structure

To end a parsing operation, close the file, and release all resources allocated for the parsing operation, call the `DeleteIFFParser()` function.

```
Err DeleteIFFParser (IFFParser *iff)
```

You pass the parser handle.

Defining, Saving, and Finding Collection Chunks

A collection chunk is just a chunk that you want the parser to save for later use. You define a particular kind of chunk as being a collection chunk by calling the `RegisterCollectionChunks()` function and specifying the chunk ID (e.g., "SSND") of the chunk along with the type field (e.g., "AIFC") of the container chunk that contains it.

When the parser encounters a chunk that has been defined as a collection chunk, the parser reads the chunk's data into a buffer and creates a `CollectionChunk` structure. The `CollectionChunk` structure points to the data buffer. See Example 16-5.

Example 16-5 *CollectionChunk structure*

```
typedef struct CollectionChunk
{
    struct CollectionChunk *cc_Next;
    ContextNode             *cc_Container;
    void                    *cc_Data;
    uint64                  cc_DataSize;
} CollectionChunk
```

The parser creates a separate linked list for each kind (i.e., type/ID combination) of collection chunk. If you are collecting MARK and SSND chunks in an AIFC FORM, for example, there will be one linked list of `CollectionChunk` structures for AIFC/MARK chunks and another for AIFC/SSND chunks. The `CollectionChunk` structure for the most recently encountered chunk is always the first one in each list.

As a typical example of when you might use collection chunks, you might define SSND chunks containing sound effects as collection chunks so that later on, you could find them and play the sound effects.

Defining Collection Chunks

To define certain kinds of chunks as collection chunks, call the function `RegisterCollectionChunks()`.

```
Err RegisterCollectionChunks(IFFParser *iff,
                             const IFFTypeID typeids[])
```

The `typeids` argument is an array of `IFFTypeID` structures, which indicate the kinds of chunks to collect. See Example 16-6 below.

Example 16-6 IFFTypeID structure

```
typedef struct IFFTypeID
{
    PackedID Type;    /* container ID (e.g.: AIFC) */
    PackedID ID;      /* chunk ID (e.g.: SSND)      */
} IFFTypeID;
```

If a `typeids` array contains a member with `Type="AIFC"` and `ID="SSND"`, for example, then the parser would collect any SSND chunks found in an AIFC FORM.

What this `RegisterCollectionChunks()` actually does is to set up a system-provided entry handler for each specified kind of chunk. The entry handler, which is invoked when that kind of chunk is encountered, reads the chunk's data into storage and creates the appropriate `CollectionChunk` structure.

The scope of the registration operation is the scope of the context node that is current when the `RegisterCollectionChunks()` function is called. The linked lists of `CollectionChunk` structures are chained to that node.

After that node is popped from the stack, that registration operation (i.e., the definition of chunks to be collected) is no longer in effect. In most cases you will want to call `RegisterCollectionChunks()` before calling `ParseIFF()` so that the scope of the registration operation will be the entire file.

The scope of a collection chunk itself, however, is the scope of the container chunk that contains it. When that container chunk's context node is popped off the stack, all the collection chunks associated with it are deleted from the collection chunk lists for the registration operation currently in effect.

Finding Collection Chunks

To find a collection chunk, call the `FindCollection()` function.

```
CollectionChunk *FindCollection(const IFFParser *iff,  
                                PackedID type, PackedID id)
```

You specify the combination of container type and chunk id to look for (e.g., "AIFC" and "SSND"). The function returns a pointer to the head of the linked list of `CollectionChunk` structures for that kind of collection chunk. This will be the most recently encountered chunk in that list.

You can then traverse the list to find other chunks of that kind.

Defining, Saving, and Finding Property Chunks

Property Chunks vs. PROP Chunks

A property chunk is any local chunk (e.g., a TXTR M2PI chunk) that you want to define as being a "property" -- i.e., an attribute -- and that you want to use according to a set of simple scoping rules.

A PROP chunk is a container chunk that appears at the beginning of a LIST chunk and contains chunks that are intended to be global within the LIST. These will often be property chunks, but they do not have to be. A chunk becomes a property chunk by being defined as one using the `RegisterPropChunks()` function, not just by being in a PROP container chunk.

The use and scoping rules for property chunks can be illustrated by the example shown in Example 16-7 below: a LIST with a nested LIST that contains FORMS.

Example 16-7 *Outline of a LIST containing another LIST, which contains FORMs*

```
LIST TXTR

    PROP TXTR
        M2TX
        M2PI

    LIST TXTR

        PROP TXTR
            M2PI

        FORM TXTR
            M2TD

        FORM TXTR
            M2PI
            M2TD

        FORM TXTR
            M2TX
            M2TD
```

In this example, each TXTR FORM needs a Header (M2TX) chunk and a Pen Index Palette (M2PI) chunk along with its Texel Data (M2TD) chunk.

If you use the RegisterPropChunks () function to define M2TX and M2PI chunks as property chunks then,

- ◆ The first TXTR FORM will use
 - ◆ The M2TX chunk from the first PROP chunk.
 - ◆ The M2PI chunk from the second PROP chunk (which overrides the M2PI chunk in the first PROP chunk).
- ◆ The second TXTR FORM will use
 - ◆ The M2TX chunk from the first PROP chunk.
 - ◆ Its own local M2PI chunk (which overrides the M2PI chunks in the first and second PROP chunks).
- ◆ The third TXTR FORM will use
 - ◆ Its own local M2TX chunk (which overrides the M2TX chunk in the first PROP chunk).
 - ◆ The M2PI chunk from the second PROP chunk (which overrides the M2PI chunk in the first PROP chunk).

The purpose of the IFF folio functions for property chunks is to automate the procedures for saving property chunks and for using them according to the proper scoping rules.

You use `RegisterPropChunks()` to define a chunk as a property chunk. You then use `FindPropChunk()` to find the appropriate property chunk to use in a particular FORM.

When the parser encounters a property chunk in a PROP chunk or a FORM chunk, it reads the property chunk's data into a buffer pointed to by a `PropChunk` structure. The parser keeps track of each chunk's precedence in the usage stack for that property.

When you call the `FindPropChunk()` function to obtain the desired chunk for a property, `FindPropChunk()` finds the appropriate saved property chunk based on the scoping rules.

Note that property chunks do not have to be in a PROP container in a LIST. They can be in a FORM that contains embedded FORMs. The scoping rules work the same way.

Property Chunks vs. Collection Chunks

When using collection chunks, you want to find a chain of collection chunks to traverse because you intend to use more than one of the same kind.

When using property chunks, you only want to find the one property chunk that contains the desired property and satisfies the scoping rules.

Defining Property Chunks

To define certain kinds of chunks as property chunks, call the function `RegisterPropChunks()`.

```
Err RegisterPropChunks(IFFParse *iff,  
    const IFFTypeID typeids[])
```

The `typeids` argument is an array of `IFFTypeID` structures, which indicate the kinds of chunks to save as property chunks.

If a `typeids` array contains a member with `Type="TXTR"` and `ID="M2PI"`, for example, then the parser saves as property chunks any M2PI chunks found in a TXTR PROP chunk or in a TXTR FORM chunk.

What this function actually does is to set up a system-provided entry handler for each specified kind of property chunk. When invoked, the entry handler creates the necessary structures and reads in the chunk's data.

The scope of the registration operation is the scope of the context node that is current when the `RegisterPropChunks()` function is called.

After that node is popped from the stack, that registration operation (i.e., the definition of chunks to be considered property chunks) is no longer in effect. In most cases you will want to call `RegisterPropChunks()` before calling `ParseIFF()` so that the scope of the registration operation will be the entire file.

The scope of a property chunk itself, however, is the scope of the LIST or FORM chunk that contains it.

Finding Property Chunks

To find a property chunk that contains a desired property and satisfies the scoping rules, call the `FindPropChunk()` function.

```
PropChunk *FindPropChunk(const IFFParser *iff,  
                          PackedID type, PackedID id)
```

You specify the combination of container type and chunk ID to look for (e.g., "TXTR" and "M2PI"). `FindPropChunk()` returns a pointer to a `PropChunk` structure that points to the data buffer for the appropriate property chunk.

Defining Stop Chunks

To tell the parser to stop and return control to your application when it encounters a particular kind of chunk, call the `RegisterStopChunks()` function.

```
Err RegisterStopChunks(IFFParser *iff,  
                       const IFFTypeID typeids[])
```

The `typeids` argument is an array of `IFFTypeID` structures, which indicate the kinds of chunks to stop for.

If a `typeids` array contains a member with `Type="TXTR"` and `ID="M2TD"`, for example, then the parser stops and returns control to your program when it encounters a M2TD chunk in a TXTR FORM.

What this function actually does is to set up a system-provided entry handler for each specified kind of stop chunk. When invoked, the entry handler creates the necessary structures and then returns control to your application.

The scope of the registration operation is the scope of the node that is current when the `RegisterStopChunks()` function is called.

After that node is popped from the stack, that registration operation (i.e., the definition of chunks to be considered stop chunks) is no longer in effect. In most cases you will want to call `RegisterStopChunks()` before calling `ParseIFF()` so that the scope of the registration operation will be the entire file.

When `ParseIFF()` encounters a stop chunk and gives control to your application, the IFF stream read cursor is positioned at the first data byte of the chunk. Your application can call `ReadChunk()` to read the chunk's data bytes

into storage. When your application has finished processing the chunk, it can call `ParseIFF()` to resume the parse operation, which will begin with the chunk immediately following the stop chunk.

Defining Entry and Exit Handlers

The `RegisterCollectionChunks()`, `RegisterPropChunks()`, and `RegisterStopChunks()` functions set up standard entry handlers that are provided by the system.

If you need more flexibility, you can set up your own entry handlers, which call routines provided by you when the specified kinds of chunks are encountered.

You can also set up exit handlers, which are invoked when leaving a specified kind of chunk just before the chunk is popped from the stack.

Defining an Entry Handler for Chunks

To set up an entry handler for a particular kind of chunk, call the `InstallEntryHandler()` function.

```
Err InstallEntryHandler(IFFParser *iff, PackedID type,
                        PackedID id, ContextInfoLocation pos, IFFCallback cb,
                        const void *userData);
```

You provide as arguments the parser handle, the container type (e.g., AIFC), the chunk ID (e.g., SSND), the context stack position, a pointer to the routine that you want the entry handler to call, and a pointer to user data that you want it to use.

The context stack position determines the scope within which the entry handler will be invoked and also when it will be popped of the stack and disposed of.

You specify one of 3 values:

- ◆ `IFF_CIL_TOP`: Attach the entry handler to the top (current) context node.
- ◆ `IFF_CIL_PROP`: Attach the entry handler to the FORM or LIST context node that is closest to the top of the context stack. This is referred to as the top property node.
- ◆ `IFF_CIL_BOTTOM`: Attach the entry handler to the context node at the bottom of the stack. The bottom context in the stack is known the “default context.” It is created by `CreateIFFParser()` as part of the initial parser structure and is not associated with a particular chunk.

(Note that these are the same values that you specify for the `StoreContextInfo()` function.)

Consider as an example a LIST that contains FORMS that contain local chunks. Assume that you call `InstallEntryHandler()` when the context stack is as shown in Example 16-8.

Example 16-8 *Top, top property, and bottom nodes in a sample context stack*

```
Context Node for local chunk M2TD    (top node)
      |
Context Node for FORM chunk TXTR     (top property node)
      |
Context Node for LIST chunk TXTR
      |
      Default Context Node    (bottom node)
```

Specifying `IFF_CIL_TOP` would probably be pointless because the entry handler would disappear as soon as the M2TD chunk is exited.

Specifying `IFF_CIL_PROP` would attach the entry handler to the FORM chunk's context. It would be invoked for any appropriate chunk contained in that FORM. It would only disappear when the end of the FORM is reached.

Specifying `IFF_CIL_BOTTOM` would attach the entry handler to the default context node. The entry handler would be invoked for any chunk of the specified type in the file. The entry handler would only disappear when `DeleteIFFParser()` deletes the parser structure.

Defining an Exit Handler for Chunks

To set up an exit handler for a particular kind of chunk, call the `InstallExitHandler()` function.

```
Err InstallExitHandler(IFFParser *iff, PackedID type,
    PackedID id, ContextInfoLocation pos, IFFCallback cb,
    const void *userData);
```

You specify the same parameters as for `InstallEntryHandler()`. The difference is that the handler is invoked when the parser exits the specified kind of chunk just before the chunk's context is popped.

Using ContextInfo Structures

The IFF folio lets you attach your own data to a context node for use when processing chunks within the scope of that context.

To accomplish this, your application

- ◆ Calls `AllocateContextInfo()` to allocate and initialize a `ContextInfo` structure that points to a buffer that will contain your data.
- ◆ Copies your data into the buffer provided by `AllocateContextInfo()`.
- ◆ Calls `StoreContextInfo()` or `AttachContextInfo()` to connect the `ContextInfo` structure to the appropriate context node.

Later on, your application can execute IFF folio functions to find the context information when it's needed.

The format of a `ContextInfo` structure is shown in Example 16-9.

Example 16-9 `ContextInfo` structure

```
typedef struct ContextInfo
{
    MinNode      ci; /* For chaining */
    uint64       ci_DataSize; /* Size of your data */
    void         *ci_Data; /* Pointer to data buffer */
} ContextInfo;
```

Allocating a ContextInfo Structure

To allocate a `ContextInfo` structure and a data buffer, call the function `AllocateContextInfo()`.

```
ContextInfo *AllocContextInfo(PackedID type, PackedID id,
                              PackedID ident, uint32 dataSize,
                              IFFCallback cb)
```

You provide as arguments

- ◆ A container type (e.g., "TXTR"), a chunk ID (e.g., "M2TD"), and an identifier of your choice.

The combination of these three arguments is what you use later on when searching for a particular `ContextInfo` structure.

You can, in fact, specify arbitrary character values in the type and ID arguments; although you would normally use meaningful values, such as the container type and local chunk ID that are particularly associated with the context information.

- ◆ The number of bytes to allocate for your application's use.
- ◆ A pointer to an optional callback function to call `FreeContextInfo()` and handle application-specific cleanup when the context node containing this structure is popped. By default, the folio just calls `FreeContextInfo()`.

`AllocateContextInfo()` allocates a zeroed-out buffer to contain your data and a `ContextInfo` structure that points to it. At this point the structure is not connected to any context node.

Inserting a ContextInfo Structure onto Context Stack

To connect the ContextInfo structure to one of three logical positions in the context stack, call the function StoreContextInfo().

```
Err StoreContextInfo(IFFParser *iff, ContextInfo *ci,  
                    ContextInfoLocation pos)
```

You provide as arguments the parser handle, a pointer to the ContextInfo structure that you previously allocated, and where to connect the structure in the context stack. Note that you can attach a ContextInfo structure to multiple context nodes.

For the stack position, you specify one of three values:

- ◆ IFF_CIL_TOP: Attach the ContextInfo structure to the top (current) context node.
- ◆ IFF_CIL_PROP: Attach the ContextInfo structure to the FORM or LIST context node that is closest to the top of the context stack. This is referred to as the top property node.
- ◆ IFF_CIL_BOTTOM: Attach the ContextInfo structure to the default context node at the bottom of the stack.

(Note that these are the same values that you specify for the InstallEntryHandler() and InstallExitHandler() functions.)

Attaching a ContextInfo Structure to a Specific ContextNode

Normally, you will use StoreContextInfo(), but occasionally, you may want to attach a ContextInfo structure to a specific context node that is not one of the three covered by StoreContextInfo().

In this case, you call the function AttachContextInfo().

```
void AttachContextInfo(IFFParser *iff, ContextNode *to,  
                      ContextInfo *ci)
```

You pass as arguments a pointer to the desired context node and a pointer to your ContextInfo structure.

To get a pointer to the desired context node, you may have to use the GetCurrentContext() and GetParentContext() functions and possibly traverse the context stack manually.

Finding a ContextInfo Structure

To search for a ContextInfo structure, call the FindContextInfo() function.

```
ContextInfo *FindContextInfo(const IFFParser *iff, PackedID type,  
                             PackedID id, PackedID ident)
```

You pass as the arguments the parser handle and the same combination of container type, chunk ID, and identifier that you passed to `AllocContextInfo()`.

`FindContextInfo()` searches back through the context stack, beginning at the top context node, and returns a pointer to the first `ContextInfo` structure having the search values that you have specified.

Removing a ContextInfo Structure from Context Stack

To remove (i.e., detach) a `ContextInfo` structure from the context node to which it is attached, call the `RemoveContextInfo()` function, passing a pointer to the `ContextInfo` structure.

```
void RemoveContextInfo(ContextInfo *ci)
```

Note that the structure and its associated data buffer are not deallocated. They still exist.

Deallocating a ContextInfo Structure

To free the memory allocated for a `ContextInfo` structure and its associated user buffer, call the `FreeContextInfo()` function.

```
void FreeContextInfo(ContextInfo *ci)
```

You pass a pointer to the `ContextInfo` structure that you want to get rid of. `FreeContextInfo()` frees both the memory allocated for the structure and the memory that `AllocContextInfo()` allocated for the user buffer.

`FreeContextInfo()` does not free any other memory that might be associated with the structure, and it does not call any custom cleanup function that you may have specified to `AllocContextInfo()`.

`FreeContextInfo()` also does not remove (i.e., detach from its context node) a `ContextInfo` structure. Be sure that the `ContextInfo` structure has already been detached before attempting to deallocate it. Otherwise you may corrupt the parsing context.

You can be sure that the `ContextInfo` structure has been detached if either of the following is true:

- ◆ You have explicitly detached it by calling the `RemoveContextInfo()` function.
- ◆ You are in the custom cleanup function that you specified to `AllocContextInfo()`. The `ContextInfo` structure will have been detached automatically at this point, but you still need to deallocate it.

Finding Context Nodes

Getting the Current ContextNode

To get a pointer to the current context node -- i.e., the context node at the top of the stack -- call the `GetCurrentContext()` function.

```
ContextNode *GetCurrentContext(const IFFParser *iff)
```

You pass a parser handle, and `GetCurrentContext()` returns a pointer to the context node most recently pushed onto that parser's context stack. This is the context node for the chunk where the stream is currently positioned.

Getting the Parent Context Node

To get a pointer to a specified context node's parent context node, call the `GetParentContext()` function.

```
ContextNode *GetParentContext(ContextNode *cn)
```

You pass a pointer to a context node, and `GetParentContext()` returns a pointer to the next context node down in the stack. This is the context node for the container chunk (e.g., FORM) that contains the chunk whose context node you specified.

Pushing and Popping Context Nodes

The parser automatically pushes and pops context nodes for the chunks it encounters during a read operation.

You do have to push and pop chunks yourself when writing an IFF file. You call `PushChunk()` to create and push onto the stack a context node for the chunk you are about to write. Then you call `WriteChunk()` to write the chunk. Then you call `PopChunk()` to pop the context node for that chunk off the stack. See "Writing out a FORM" on page 254.

Pushing a New Context Node onto the Context Stack

To create a new context node and push it onto the context stack, call the `PushChunk()` function. The IFF folio uses this function internally to read and write chunks, but your application would normally use it only to write.

```
Err PushChunk(IFFParser *iff, PackedID type, PackedID id,  
              uint32 size)
```

You pass as parameters the parser handle, the container type (e.g., TXTR), the chunk ID of the chunk being read or written (e.g., M2TD), and the number of data bytes in the chunk. For read operations, you read this information from the chunk header. For write operations, you get it from memory.

If you are writing and don't know the number of bytes yet, then specify `IFF_SIZE_UNKNOWN_32` (if you know that the size will fit in 32 bits) or `IFF_SIZE_UNKNOWN_64` (if you know it won't).

If you specify a specific size for the chunk, the parser will enforce it and return an error if you try to write more. Otherwise the parser will just keep track of the number of bytes that are written.

Popping the Top Context Node off the Context Stack

To pop the top context node off the stack and free all associated `ContextInfo` structures, call the `PopChunk()` function.

```
Err PopChunk(IFFParser *iff)
```

Reading and Writing Chunks

Reading a Chunk into a Buffer

To read bytes from the data portion of a chunk into a buffer, call the `ReadChunk()` function.

```
int32 ReadChunk(IFFParser *iff, void *buffer, uint32 numBytes)
```

You pass as arguments the parser handle, a pointer to the buffer in which to put the data, and the number of bytes to read into the buffer.

`ReadChunk()` begins reading at the current read cursor position, which may be past the chunk's first data byte because of a previous `ReadChunk()` operation.

`ReadChunk()` reads until it has read the specified number of bytes or filled the buffer or reached the end of the chunk.

`ReadChunk()` then returns the number of bytes actually read (or a negative error code). If you try to read past the end of the chunk, `ReadChunk()` reads to the end of the chunk and then returns with the number of bytes actually read.

Writing a Chunk to a File

To write data from a buffer to the current chunk in an IFF file, call the `WriteChunk()` function.

```
int32 WriteChunk(IFFParser *iff, const void *buffer,
                uint32 numBytes)
```

You pass as arguments the parser handle, a pointer to the buffer containing the data to write, and the number of bytes to write. Note that you must first have called `PushChunk()`.

`WriteChunk()` begins writing at the current write position in the file, which may be past the chunk's first data byte because of a previous `WriteChunk()` operation.

`WriteChunk()` writes until it has written the specified number of bytes or emptied the buffer or reached the maximum chunk size as specified in the `PushChunk()` call.

`WriteChunk()` then returns the number of bytes actually written (or a negative error code).

If you specified `IFF_SIZE_UNKNOWN_32` or `IFF_SIZE_UNKNOWN_64` in the `PushChunk()` call, then the parser keeps track of the total number of bytes written to the chunk, and the `PopChunk()` operation fills in the chunk's size field.

Positioning the Cursor in a Stream

Moving the Position Cursor within a Chunk

To move the current position cursor within the current chunk, call the `SeekChunk()` function.

```
Err SeekChunk(IFFParser *iff, int32 position,
              IFFSeekModes mode)
```

The position cursor is the byte position at which the next read or write will start in the chunk.

You pass as arguments the parser handle, the desired position of the cursor, and a value indicating whether that position is relative to the beginning of the chunk, relative to the current position of the cursor, or relative to the end of the chunk.

Getting the Seek Position in the IFF Stream

To get the absolute seek position, which is the number of bytes from the start of the IFF stream to the current position cursor, call the `GetIFFOffset()` function.

```
int64 GetIFFOffset(IFFParser *iff)
```

You pass the parser handle, and the function returns the absolute seek position. You might use this to determine the exact position of a chunk or part of a chunk in an IFF file.

The Icon Folio

This chapter describes the Icon folio, which facilitates saving an icon to a file and loading an icon into memory.

This chapter contains the following topics:

Topic	Page Number
Introduction - What the Icon Folio Does	271
Loading an Icon into Memory	272
Unloading an Icon from Memory	274
Saving an Icon	274

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Introduction - What the Icon Folio Does

About Icons

An icon is a graphical element that your application displays as a symbol for an element handled by your application, such as a file, a file system, a device, or an operation. Your application may create and store the icon, or it may just display an icon that was created and stored by another application.

The components of an icon are

- ◆ A set of one or more sprite objects to be displayed.

An icon might have multiple sprites to represent multiple states of an element. For example, one sprite might represent the element before being selected, and another sprite might represent the element after being selected.

Or an icon might have multiple sprites to provide animation. In this case, the time to wait between frames will be stored along with the sprites.

- ◆ Information about the icon, such as the name of the application that created it and the time to wait between frames when displaying each frame of an animated multi-sprite object.

All icons used in the 3DO M2 system are stored as IFF "FORM ICON" chunks. IFF is a file formatting and parsing system used by M2 to facilitate retrieving and parsing stored information. If you are not already familiar with IFF, see Chapter 16, "The IFF Folio".

An icon may be stored in one of the following ways.

- ◆ In single "FORM ICON" chunk in a stand-alone file.
- ◆ In a "FORM ICON" chunk imbedded within a another IFF file that serves some larger purpose.
- ◆ In the system data associated with a file system. This would be the case for an icon whose purpose is to represent that file system.
- ◆ In the system data associated with a device. This would be the case for an icon whose purpose is to represent that device.

Functions Provided by the Icon Folio

The Icon folio provides the following functions to manage the details involved in loading and saving icons:

- ◆ The `LoadIcon()` function loads an icon into memory. It can load an icon from a stand-alone "FORM ICON" file, from an imbedded "FORM ICON" chunk in a larger IFF file, from a file system, or from a device driver. The icon's data is parsed and organized into an in-memory structure where it can be displayed by your application.
- ◆ The `UnloadIcon()` function deletes an icon's in-memory data structure from memory and frees resources associated with the icon.
- ◆ The `SaveIcon()` function stores an icon as a stand-alone "FORM ICON" file or as an imbedded "FORM ICON" chunk in a larger IFF file. You provide pointers to the sprite(s) and related data, and `SaveIcon()` organizes the data and writes it out in IFF format.

Loading an Icon into Memory

To load an icon into memory, use the `LoadIcon()` function.

```
Err LoadIcon(Icon **icon, const TagArg *tags)
```

You provide as arguments

- ◆ A pointer to the location in which to put the pointer to the icon.
- ◆ An array of tag arguments that tell the function where to get the icon.

You must specify one of the following tags. Note that they are mutually exclusive.

- ◆ To load an icon from a stand-alone "FORM ICON" file, use the tag `LOADICON_TAG_FILENAME` `-(char *)`, with a pointer to a character string containing the file's pathname (relative or absolute).
- ◆ To load an icon from a larger IFF file in which it is imbedded, use the tag `LOADICON_TAG_IFFPARSER` `(IFFParser *)`, with a pointer to a the IFF parsing structure set up to control the parsing of the IFF file.

When you use this tag, you must also use the tag `LOADICON_TAG_IFFPARSETYPE` `(uint32)` to indicate the current parsing status for the file.

A value of `LOADICON_TYPE_AUTOPARSE` tells `LoadIcon()` to look for a "FORM ICON" chunk in the file.

A value of `LOADICON_TYPE_PARSED` tells `LoadIcon()` that the caller has already parsed up to and recognized a "FORM ICON" chunk and that `LoadIcon()` should parse from that point in the IFF stream.

- ◆ To load an icon representing a file system from the system data associated with the file system, use the tag `LOADICON_TAG_FILESYSTEM` `(char *)`, with a pointer to a character string containing the filesystem's absolute pathname.

The icon for a filesystem may be stored in several different places. `LoadIcon()` knows where to look for it and searches in each of the possible locations.

- ◆ To load an icon representing a device from the system data associated with the device, use the tag `LOADICON_TAG_DRIVER` `(char *)`, with a pointer to a character string containing the name of the device driver.

`LoadIcon()` retrieves the icon from the device driver, usually without loading the driver itself into memory.

`LoadIcon()` parses the file containing the icon data and loads the information into the in-memory structure shown in the example below:

Example 17-1 *In-memory representation of an icon*

```
typedef struct Icon
{
    Node Node; /* For users to use as they wish */
    List SpriteObjs; /* A list of SpriteObjs with imagery */
    TimeVal TimeBetweenFrames; /*Opt. time between frames */
    char ApplicationName[32]; /* Name of App that created icon*/
} Icon;
```

Unloading an Icon from Memory

To delete an icon from memory and free any resources allocated by `LoadIcon()` for it, use the `UnloadIcon()` function.

```
Err UnloadIcon(Icon *icon)
```

You provide as an argument a pointer to the icon's `Icon` structure in memory.

Saving an Icon

To save an icon in a stand-alone "FORM ICON" file or in a "FORM ICON" chunk inside another IFF file, use the `SaveIcon()` function.

```
Err SaveIcon(char *UTFfilename, char *AppName,
             const TagArg *tags)
```

You specify as arguments,

- ◆ The pathname (relative or absolute) of the file containing the sprite(s) for the icon.
- ◆ The name of the application that created the icon.
- ◆ A pointer to an array of tag arguments that contain information about the icon and tell the function where to store it.
 - ◆ To save the icon in a stand-alone "FORM ICON" file, specify the tag `SAVEICON_TAG_FILENAME` (`char *`) with the pathname (relative or absolute) of the target file.
 - ◆ To imbed the icon as a "FORM ICON" chunk in another IFF file, specify the tag `SAVEICON_TAG_IFFPARSER` (`IFFParser *`) with a pointer to the IFF parsing structure for the target file.
 - ◆ To store the amount of time to wait between frames when displaying an animated icon, specify the tag `SAVEICON_TAG_TIMEBETWEENFRAMES` (`TimeVal *`) with the desired time interval in seconds and microseconds.

The SaveGame Folio

This chapter describes the SaveGame folio, which lets you save game data to a file or load saved game data from a file into memory.

This chapter contains the following topics:

Topic	Page Number
Introduction - What the SaveGame Folio Does	275
Saving Game Data	276
Loading Saved Game Data into Memory	278

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

Introduction - What the SaveGame Folio Does

What Kind of Data is Saved - Determined by Application

The type of data saved is determined by your application. You may want to save

- ◆ The current state of a game, so that the end-user can resume playing later.
- ◆ A table of scores or game levels achieved by end-users.
- ◆ Options or preferences set by end-users.
- ◆ Some other type of game data.

Your application determines the kind of data to save and provides the data to the SaveGame folio. The purpose of the SaveGame folio is to simplify the process of saving and reloading game data by managing some of the details involved. The SaveGame folio also handles data compression and decompression automatically.

Saved game data is stored as a "FORM SGME" chunk in an IFF-formatted file. A "FORM SGME" chunk can be a stand-alone chunk in its own file, or it can be a sub-chunk embedded in a larger IFF file. IFF is a file formatting convention used by 3DO to facilitate retrieving and parsing stored information. If you are not already familiar with IFF, see Chapter 16, "The IFF Folio".

Functions Provided by the SaveGame Folio

The SaveGame folio provides the following functions to save and load game data:

- ◆ The `SaveGameData()` function compresses the game data (to save filesystem space) and stores it into a file in IFF format.
You provide a structure or array of structures pointing to the data to be saved.
You can save the data into a stand-alone saved-game file containing a single "FORM SGME" chunk, or you can save it as a "FORM SGME" chunk embedded in another IFF file.
You can also provide icon information and textures, which `SaveGameData()` will save as a "FORM ICON" chunk inside the "FORM SGME" chunk.
- ◆ The `LoadGameData()` function reads saved game data from a file, decompresses it, and loads it into buffer storage pointed to by a structure or an array of structures.
You can load game data from a single "FORM SGME" chunk in a stand-alone file or from a "FORM SGME" chunk embedded in another IFF file.
If the "FORM SGME" chunk contains a "FORM ICON" chunk, `LoadGameData()` will load the icon data into an `Icon` structure and give you a pointer to the structure.

Saving Game Data

To save game data, call the `SaveGameData()` function.

```
Err SaveGameData(const char *appname, const TagArg *tags)
```

You provide as arguments

- ◆ A pointer to a character string containing the name of the application that created the saved data.
- ◆ A pointer to an array of tag arguments that provide the following information:
Where to store the data.

- ◆ To save the data in a stand-alone “FORM SGME” file, specify the tag `SAVEGAMEDATA_TAG_FILENAME (const char *)`, specifying a pointer to the pathname (relative or absolute) of the target file.
- ◆ To embed the saved data as a “FORM SGME” chunk in another IFF file, specify the tag `SAVEGAMEDATA_TAG_IFFPARSER (IFFParser *)`, specifying a pointer to the IFF parsing structure for the target file. `SaveGameData()` will write the data beginning at the location in the IFF stream currently pointed to by the IFF parser.

How your application will provide the data to be saved.

- ◆ Your application can provide an array of `SGData` structures, each of which points to a buffer containing a chunk of data to be saved and also specifies the length of the data. `SaveGameData()` processes the buffers in sequential order.

To provide the data this way, specify the tag `SAVEGAMEDATA_TAG_BUFFERARRAY (SGData *)`, specifying a pointer to the array. See Example 18-1 on page 278 for the format of the `SGData` structure.

- ◆ Your application can provide a callback function that `SaveGameData()` calls iteratively to get each chunk of data. `SaveGameData()` passes your callback function a pointer to an `SGData` structure, which your callback function fills in with the length of the data and a pointer to the data.

To provide the data this way, specify the tag `SAVEGAMEDATA_TAG_CALLBACK (GSCallBack *)`, specifying a pointer to the your callback function.

You can also include the tag `SAVEGAMEDATA_TAG_CALLBACKDATA (void *)`, specifying a pointer to additional application-specific data that you want your callback function to be passed each time it is called. This can be context data, e.g., non-global application variables that you want to reference, or any other data that you desire.

Icon data if you want to include an icon in the saved game data.

- ◆ Use the tag `SAVEGAMEDATA_TAG_ICON (const char *)` to provide the pathname (relative or absolute) of a UTF file containing one or more icon sprites to be saved in a “FORM ICON” chunk within the “FORM SGME” chunk.
- ◆ For animated multi-sprite icons, specify the additional tag `SAVEGAMEDATA_TAG_TIMEBETWEENFRAMES (TimeVal *)` to provide the interval, in seconds and microseconds, to wait between each frame when displaying the icon.

Other data:

- ◆ To provide a string that identifies this particular saved game data (e.g., “Bob on Level 3”, specify the tag `SAVEGAMEDATA_TAG_IDSTRING (char *)`, specifying a pointer to the string.

Note that you just provide the data that you want to save. You do not need to provide any IFF keywords or formatting.

`SaveGameData()` returns a negative error code if it fails to write out all the data that you provide.

Example 18-1 *The SGDATA structure*

```
typedef struct SGData
{
    void          *Buffer; /* Points chunk of game data */
    uint32        Length; /* Bytes to be read or written */
    uint32        Actual; /* Bytes actually read (for load)*/
} SGData;
```

Caution: Each `SaveGameData()` call causes a write operation to the microcard. Remember that you want to avoid unnecessary write operations to microcards. See "Important Note About Microcard File Systems" on page 123.

Loading Saved Game Data into Memory

To load saved game data into memory, call the `LoadGameData()` function.

```
Err LoadGameData(LoadedGame *game, const TagArg *tags)
```

You provide as an argument a pointer to an array of tag arguments that provide the following information:

- ◆ Where to get the saved data.
 - ◆ To read the saved data from a stand-alone "FORM SGME" file, specify the tag `LOADGAMEDATA_TAG_FILENAME` (`const char *`), specifying a pointer to the pathname (relative or absolute) of the file containing the saved data.
 - ◆ To read an embedded "FORM SGME" chunk from within a larger IFF file, specify the tag `LOADGAMEDATA_TAG_IFFPARSER` (`IFFParser *`), specifying a pointer to the IFF parsing structure for the file containing the saved data.
- ◆ How to give the data to your application.
 - ◆ Your application can provide an array of `SGData` structures, each of which points to a buffer to be filled with a chunk of data. `LoadGameData()` will load the buffers in order and fill in the "Actual" field with the number of bytes actually loaded.

To load the data this way, specify the tag `LOADGAMEDATA_TAG_BUFFERARRAY (SGData *)`, specifying a pointer to the array.

- ◆ Your application can provide a callback function that `LoadGameData()` calls iteratively to provide each chunk of data.

`LoadGameData()` passes your callback function a pointer to an `SGData` structure containing the size of the chunk to be loaded. If your application wants to load the chunk, it fills in the buffer field with a buffer pointer and then returns. `LoadGameData()` then loads the chunk into the buffer. Note that your callback function can also indicate that it wants to skip that chunk or to terminate `LoadGameData()`.

To get the data using a callback function, specify the tag `LOADGAMEDATA_TAG_CALLBACK (GSCallBack *)`, specifying a pointer to the your callback function.

You can also include the tag `LOADGAMEDATA_TAG_CALLBACKDATA (void *)`, specifying a pointer to additional application-specific data that you want your callback function to be passed each time it is called. This can be context data, e.g., non-global application variables that you want to reference, or any other data that you desire.

- ◆ Where to put a pointer to icon data extracted from the file.
 - ◆ To request that any icon in the saved game data be loaded into an in-memory `Icon` structure, specify the tag `LOADGAMEDATA_TAG_ICON (Icon **)`, specifying a pointer to the location in which to store the pointer to the `Icon` structure when the icon has been loaded.
- ◆ Other data:
 - ◆ To get the id string for this particular saved game data (e.g., "Bob on Level 3"), specify the tag `LOADGAMEDATA_TAG_IDSTRING (char *)`, specifying a pointer to the string buffer (at least 32 bytes long) into which to copy the string.

The Requestor Folio

This chapter describes the Requestor folio, which provides a ready-made graphical user interface for interacting with 3DO file systems.

This chapter contains the following topics:

Topic	Page Number
Introduction - What the Requestor Folio Does	281
Creating a Requestor Object	283
Displaying the Storage Requestor Interface to the User	285
Querying a Storage Requestor Object	285
Modifying a Storage Requestor Object	286
Deleting a Storage Requestor Object	286

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

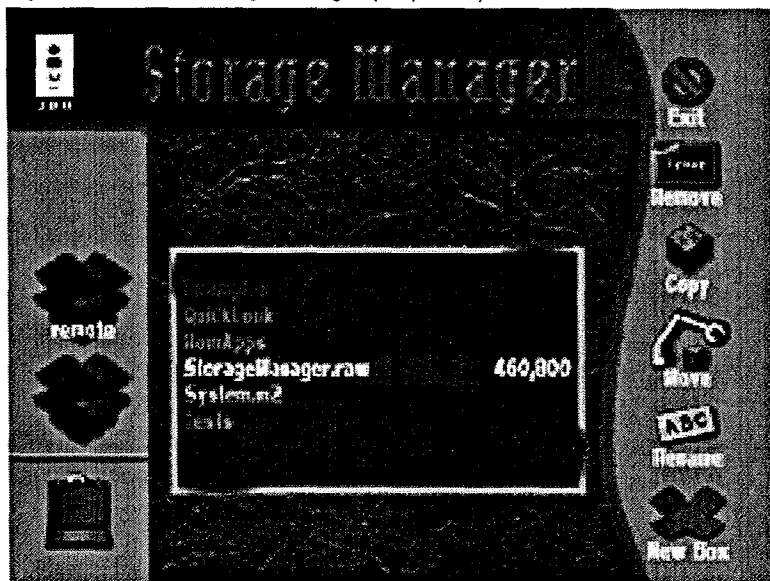
Introduction - What the Requestor Folio Does

Purpose

The Requestor folio makes it easy for you to provide your end-users with a graphical interface to interact with 3DO file systems. The Requestor folio provides a ready-to-display GUI, called the Storage Manager interface, with a file browser

and appropriate buttons, such as "OK", "Load", "Save", "Cancel", "Exit", and "Quit" buttons. Figure 19-1 below shows what the Storage Manager interface looks like.

Figure 19-1 *The Storage Manager (Requestor) Interface*



You can specify tag arguments in the Requestor folio function calls to customize the GUI, controlling which buttons are displayed, which operations end-users can perform (e.g., deleting files), and other features.

A typical situation in which you might use the Requestor folio would be if your application allows the end-user to load saved game data from a microcard. When ready to load the saved game data, your application could display the Storage Manager interface to let the user select the microcard file containing the saved data.

Note that you do not have to use the Requestor folio. You can provide your own user interface. The purpose of the Requestor folio is simply to give you a ready-made but customizable solution to save you development time.

How to Use the Requestor Folio - Overview

To let an end-user load a saved game, for example, you could follow a scenario like this:

1. The end-user selects "Load Saved Game" from your application's user interface.

2. Your application calls the `CreateStorageReq()` function to create a requestor object. The requestor object is the engine that controls the Storage Manager interface.

In the `CreateStorageReq()` function call, you specify tag arguments to define the following:

- ◆ The initial directory to display in the file browser when the Storage Manager interface comes up.
 - ◆ The initial default file to select in the directory.
 - ◆ The prompt to display, telling the end-user to select a file and click the "Load" button.
 - ◆ Which types of files to display in the file browser -- e.g., only saved-game files.
 - ◆ Which buttons to provide -- e.g., "OK", "Load", "Cancel", and "Exit" -- and which operations the user can perform -- e.g., change directories (navigate), but NOT create, delete, move, copy, or rename files and directories.
 - ◆ Some other characteristics, such as the text font used in the display.
3. Then your application calls the `DisplayStorageReq()` function, which displays the Storage Manager interface that you defined when you created the requestor object.
 4. The end-user navigates through directories on the appropriate microcard, selects the desired saved-game file, and clicks the "Load" button in the Storage Manager interface.

At this point, `DisplayStorageReq()` returns with a return code indicating that the end-user has made a choice and wants to continue. The Storage Manager interface is no longer displayed.
 5. Your application calls the `QueryStorageReq()` function, specifying tags to ask the requestor object for the directory and file that the user selected.
 6. Then your application calls the `DeleteStorageReq()` function to terminate use of the requestor object and free its resources.
 7. Finally, your application calls the `LoadGameData()` function (part of the SaveGame folio) to load the saved game file selected by the end-user.

Creating a Requestor Object

To create a requestor object, use the `CreateStorageReq()` function.

```
Err CreateStorageReq(StorageReq **req, const TagArg *tags)
```

You pass as arguments

- ◆ A pointer to a location where the function will put a pointer to the requestor object.
- ◆ An array of tag arguments that define aspects of the interface to be presented to the end-user.
 - ◆ `STORREQ_TAG_DIRECTORY (const char *)` specifies directory to display in the file browser when the Storage Manager interface first comes up. This will be considered the "current" directory, which the end-user can then change by navigating and selecting.
 - ◆ `STORREQ_TAG_FILE (const char *)` specifies the initial file to select when the display first comes up. The end-user can then navigate and select another file.
 - ◆ `STORREQ_TAG_PROMPT (SpriteObj *)` specifies the sprite to display as a prompt. This sprite usually contains instructions for the end-user.
 - ◆ `STORREQ_TAG_FILTERFUNC (FileFilterFunc)` specifies a function, provided by your application, that is called for every file scanned to determine whether that file should be displayed in the file browser.
 - ◆ `STORREQ_TAG_OPTIONS (uint32)` specifies a bitmask that determines the features to provide in the display and the operations the user can perform. The option constants are listed below.

Example 19-1 *Option flags for use with the `STORREQ_TAG_OPTIONS` tag*

```
STORREQ_OPTION_OK           /* put up an "OK" button           */
STORREQ_OPTION_LOAD         /* put up a "Load" button         */
STORREQ_OPTION_SAVE         /* put up a "Save" button         */
STORREQ_OPTION_CANCEL       /* put up a "Cancel" button       */
STORREQ_OPTION_EXIT         /* put up an "Exit" button        */
STORREQ_OPTION_QUIT         /* put up a "Quit" button         */
STORREQ_OPTION_DELETE       /* allow the user to delete files  */
STORREQ_OPTION_MOVE         /* allow the user to move files    */
STORREQ_OPTION_COPY         /* allow the user to copy files    */
STORREQ_OPTION_RENAME       /* allow the user to rename files  */
STORREQ_OPTION_CREATEDIR    /* allow the user to create dirs   */
STORREQ_OPTION_CHANGEDIR    /* allow the user to navigate      */
```

- ◆ `STORREQ_TAG_VIEWLIST (Item)` specifies the view list in which to put Storage Manager display.
- ◆ `STORREQ_TAG_FONT (Item)` specifies the font to use for text in the display. Note that, to save memory, the display is 640x240 pixels so fonts with a standard aspect of 1:1 will appear distorted. Use fonts with a 2:1 aspect ratio.

- ◆ `STORREQ_TAG_LOCALE (Item)` specifies the geographical locale to use for language and number formatting.

The `CreateStorageReq()` function can take as long as 1 or 2 seconds to complete because it must load graphics. You may want to execute this function in the background while doing something else and then call `DisplayStorageReq()`, which brings up the display immediately once the object has been created.

Displaying the Storage Requestor Interface to the User

To bring up the Storage Manager display on the screen, call the `DisplayStorageReq()` function.

```
Err DisplayStorageReq(StorageReq *req)
```

You pass as an argument a pointer to the requestor object. `DisplayStorageReq()` presents the display to the user, who can then interact with it. When the user completes the interaction, by making a choice or by cancelling the operation, `DisplayStorageReq()` returns.

Your application then examines the return code, which indicates that the user made a choice and your application should act on the choice (`STORREQ_OK`) or that the user cancelled the operation and your application should discontinue it (`STORREQ_CANCEL`).

Querying a Storage Requestor Object

After the display function returns, your application can call the `QueryStorageReq()` function to determine the current settings of the storage requestor object's attributes, which will now reflect any changes made during the end-user's interaction.

```
Err QueryStorageReq(StorageReq *req, const TagArg *tags)
```

You pass as an arguments a pointer to the storage requestor object and an array of tag arguments for the attributes that you want to query.

To determine the requestor object's current directory and file, specify the `STORREQ_TAG_DIRECTORY (char *)` and `STORREQ_TAG_FILE (char *)` tags, providing, in each case, a pointer to a string buffer. Together, these will comprise the pathname of the file that the end-user selected.

You can also find out the values of the requestor object's other attributes by using the appropriate tags.

Examples:

```
STORREQ_TAG_PROMPT (SpriteObj *)
STORREQ_TAG_FILTERFUNC (FileFilterFunc *)
STORREQ_TAG_OPTION (uint32)
STORREQ_TAG_VIEWLIST (Item *)
STORREQ_TAG_FONT (Item *)
STORREQ_TAG_LOCALE (Item *)
```

Modifying a Storage Requestor Object

You can display the same requestor object multiple times and modify the object in between display operations. To modify the attributes of a requestor object, call the `ModifyStorageReq()` function.

```
Err ModifyStorageReq(StorageReq *req, const TagArg *tags)
```

You pass as arguments a pointer to the storage requestor object and an array of tag arguments for the attributes that you want to modify.

Examples:

```
STORREQ_TAG_DIRECTORY (const char *)
STORREQ_TAG_FILE (const char *)
STORREQ_TAG_PROMPT (SpriteObj *)
STORREQ_TAG_FILTERFUNC (FileFilterFunc)
STORREQ_TAG_OPTIONS (uint32)
STORREQ_TAG_VIEWLIST (Item)
STORREQ_TAG_FONT (Item)
STORREQ_TAG_LOCALE (Item)
```

Deleting a Storage Requestor Object

When your application has finished using a requestor object, call the `DeleteStorageReq()` function to delete it.

```
Err DeleteStorageReq(StorageReq *req)
```

You pass as an argument pointer to the requestor object to be deleted.

Dynamic-Link Libraries

This chapter describes Dynamic-link libraries (DLLs), which are executable object-code libraries that are loaded dynamically (at run time) instead of statically (at link time).

This chapter contains the following topics.

Topic	Page Number
About DLLs	287
Creating and Using 3DO M2 DLLs	288
Definition Files	290
Building a Multi-Module Application	291
Functions for Handling Modules	296
Example: Creating and Using a DLL	300

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About DLLs

When portions of a program are implemented as DLLs, those portions can be loaded into memory when they are needed and then unloaded from memory when they are no longer required. Then the memory that they previously occupied can be used for other purposes.

Because DLLs allow different functions to use the same memory locations at different times, they can provide additional memory space for memory-intensive operations. That's because operations that require large amounts of memory can obtain the memory they need by unloading DLLs that are not currently needed. Subsequently, if a DLL that has been unloaded is needed again, it can be loaded back into memory—at some other memory location, if that is what is required.

Besides conserving memory, DLLs can help you make large numbers of resources available on a CD-ROM, ready to be instantly loaded into memory whenever they are needed.

DLLs can be very useful in developing game titles. When a game contains a set of procedures that do not have to be memory-resident at all times—for example, a set of procedures that lists high scores or saves a game on a micro card—that part of the program can be implemented as a DLL. Then it can be loaded into memory whenever it is needed, and unloaded when it is not needed to increase the memory that is available for the rest of the game.

The 3DO M2 Kernel offers a number of functions that you can call to create, manage, execute, and delete DLLs. This chapter describes those functions. It also presents an example tutorial that explains how to create and use a DLL.

Creating and Using 3DO M2 DLLs

If you are familiar with the way DLLs are implemented in other operating environments—for example, in MS-DOS and Windows—you will probably find many similarities in the DLL implementations you are familiar with and the way in which DLLs are implemented in M2. However, M2 DLLs also have some unique features of their own.

For example, in M2, there is no difference between the architecture of a DLL and the architecture of any other kind of executable. To implement an M2 executable as a DLL, all you have to do is define it as a DDL in a special kind of file known as a definition file (see “Definition Files” on page 290). Then you can use it as a DLL in your application.

Because any executable can be used as a DLL in an M2 application, M2 defines no special syntax for DLL entry points or DLL exit points; even an ordinary `main()` function can serve as an entry point in an M2 DLL.

Modules

Because there is no architectural difference between an M2 DLL and any other M2 executable, 3DO developers often use the same word—*module*—to refer to M2 DLLs and other kinds of M2 executables.

When functions and variables are implemented in a DLL module, other modules can access those functions and variables by *importing* them. Similarly, a DLL that implements functions and variables can make them available to other modules by *exporting* them to the modules that need to access to them.

DLLs can export functions and variables to applications, statically linked libraries, or other DLLs. Functions exported by DLLs can be imported by applications, statically linked libraries or other DLLs.

Linking Multi-Module Applications

In M2, a module can export functions and variables to other modules in several ways. The most straightforward way to export functions and variable is to use symbols. For example, if a module exports a function named `shoot ()` using a symbol, the symbol that is exported is the word `shoot`. Similarly, if a module exports the value of a variable named `x` as a symbol, the symbol that is exported is the letter `x`.

When you create an application module that imports symbols from other modules, you build the application in the usual way. First, you separately compile each of the source files that make up your application. When you have compiled everything, the result is a separate object file (.O file) for each source file. Finally, you must use the M2 linker to link each of your applications object (.O) files into an executable.

If your application makes use of the dynamic loading and unloading capabilities of DLLs, the final result of the build process is an executable application that can dynamically load functions and variables from other modules that have also been compiled and linked into executables. Consequently, a multi-module M2 application is not shipped as a single executable. Instead, it is made up of multiple executables—one executable for each of its modules.

When you ship a multi-module application that makes use of the dynamic-loading capabilities of DLLs, you must provide the user not only with your application's main executable file, but also with the executable files that the M2 linker has generated for all the other modules that your application relies upon.

Definition Files

During each build process, the M2 linker uses a special kind of text file called a *definition file* to provide your application module with the information it needs to import symbols from other modules—and to provide your application's exporting modules with the information that they need to export symbols to other modules.

In M2, definition files always have a ".X" filename extension. For example, the M2 linker recognizes a file named

```
LINKEXEC.X
```

as a definition file.

When you create a multi-module application that makes use of dynamic linking, you must provide the M2 linker with a definition file that defines all the symbols that are exported and imported by each module in your application. At link time, the M2 linker uses the information in your application's definition file to sort out the dependencies that exist among your application's various modules, and to provide each module in your application with all the information it needs to export and import any symbols that your application implements in DLLs.

How Definition Files Work

The main job of a definition file is to define the symbols that are exported and imported by the modules that make up a multi-module application. At link time, the M2 linker uses the symbol definitions provided by your application's definition file to resolve symbols that are exported and imported by the modules that make up your application. When the linker has finished this job, it translates the symbols defined in your application's definition file into binary data and places that data in the individual object file that it creates for each module used by your application. Subsequently, when your application is executed, the symbol-definition information that the linker has placed in each object file is used to manage the dynamic loading of functions and variables that are implemented in DLLs.

Example of a Definition File

Example 20-1 illustrates the format of an M2 definition file that could be used to export functions and variables.

Example 20-1 *Format of a Definition File*

```

MODULE          16384
EXPORTER        0 = fish
EXPORTER        1 = turtle
EXPORTER        3 = x

```

In the definition file shown in Example 20-1:

- ◆ The first line contains a unique 32-bit ID number that distinguishes this definition file from all other definition files. Because there is no way of knowing in advance which modules will be used together in the same application, the M2 linker may need this ID number to resolve filename conflicts in modules. The higher 16 bits of a definition file's ID number should be a developer number assigned by 3DO. The lower 16 bits should be a unique library number assigned by your site.
- ◆ The second line of the example exports a symbol for a function named `fish()`. It also assigns an identifier number called an *ordinal number* (in this case, 0) to the `fish()` function. The ordinal number that is assigned to each exported symbol in a module is what enables the loader to identify this symbol being exported.
- ◆ The third line of the example exports a function named `turtle()` and assigns the ordinal number 1 to the `turtle()` function.
- ◆ The fourth line exports a variable named `x` and assigns the ordinal number 1 to the `x` variable.

A sample program named `EXPORTER`, presented at the end of this chapter, shows how you can use a definition file to define the exports used in an M2 application.

Building a Multi-Module Application

When you have created a module that exports functions, variables, or both, you can link it to your application in two different ways:

- ◆ If you link the module to your application using the `IMPORT_NOW` flag (see Example 20-2), the M2 linker compiles the module as a DLL that loads as soon as your application starts.

- ◆ If you want M2 to treat a module as a DLL, so you load it and unload whenever you like, you must override the `IMPORT_NOW` flag with one of the other flags shown in Example 20-2. You must also define your module as a DLL by creating a special kind of text file called a definition file. To learn how to write and use definition files, see "Definition Files" on page 290.

Example 20-2 *Flags used to Link Modules*

```
IMPORT_NOW - load the DLL at the time the executable is loaded
              (import at load-time)
REIMPORT_ALLOWED - allow the DLL to be reloaded
IMPORT_ON_DEMAND - wait to load the DLL until a system call is
                  issued from the executable to do so (import at run-time)
IMPORT_FLAG <number> - arbitrary flag number (for future use)
```

Note: *You can use the `REIMPORT_ALLOWED` and `IMPORT_ON_DEMAND` flags in combination.*

Files Used in a Multi-Module Application

To build a multi-module M2 application that makes use of the dynamic-loading capabilities of DLLs, you must provide a separate object file for each module that exports or imports symbols dynamically. You must also provide a definition file that contains the definitions of all functions that will be exported or imported dynamically when your application is executed.

The sample program named `IMPORTER`, presented at the end of this chapter, shows how you can write an M2 program that makes use of DLLs. The source files used in the `IMPORTER` program are:

- ◆ `EXPORTER.C`—A source file that creates an exporting module implemented as a DLL.
- ◆ `EXPORTER.H`—A header file that goes with the `EXPORTER.C` file.
- ◆ `IMPORTER.C`—A source file that creates an importing module. In this example, the `IMPORTER.C` file is the source file for the `IMPORTER` application, which imports symbols from the `EXPORTER` DLL.
- ◆ `EXPORTER.X`—The `IMPORTER` application's definition file.
- ◆ `EXPORTER.MAKE`—An MPW-generated makefile that builds the `EXPORTER` DLL.
- ◆ `IMPORTER.MAKE`—An MPW-generated makefile that builds the `IMPORTER` application.

As you can see, the IMPORTER program has two C-language source files: one named EXPORTER.C and one named IMPORTER.C. The EXPORTER.C file is the source file for a DLL module, and the IMPORTER.C file is the source file for an application that imports symbols from the EXPORTER DLL.

There is also one header file: EXPORTER.H, which declares a structure and two functions that are used in the EXPORTER.C file.

To build the complete program, you must build two executables: the EXPORTER DLL and the IMPORTER application. Both these executables rely on the EXPORTER.X definition file, which defines the symbols exported by the EXPORTER DLL.

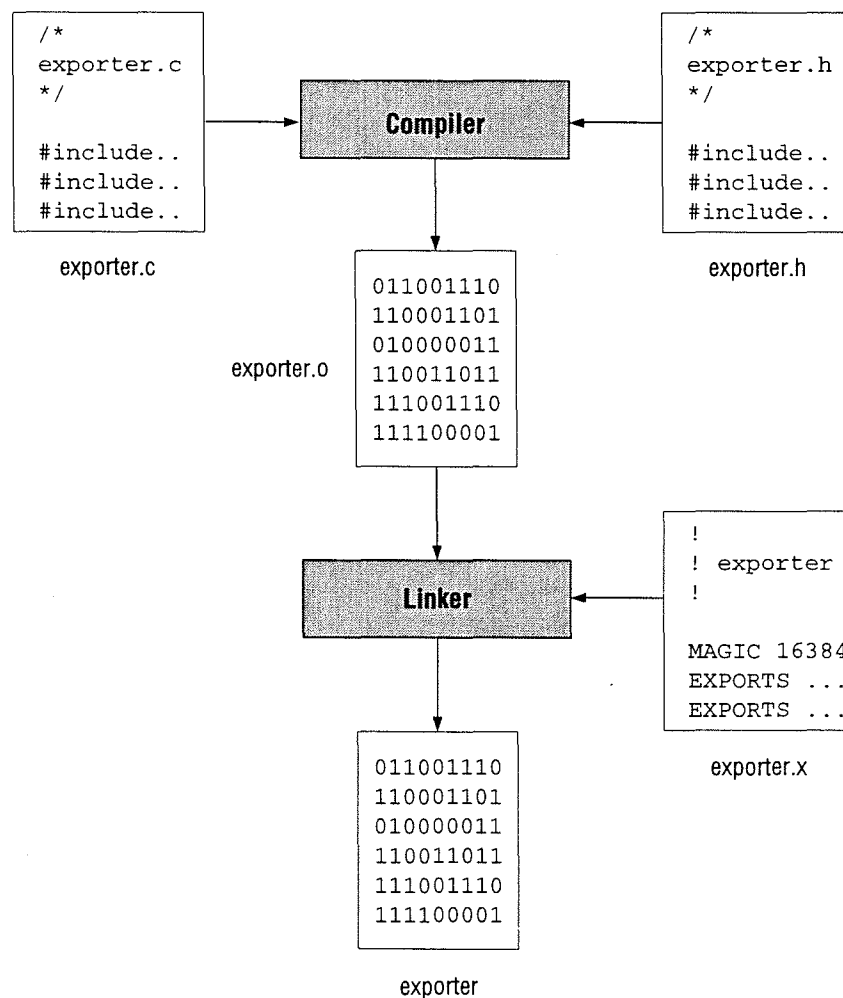


Figure 20-1 Building an exporting module

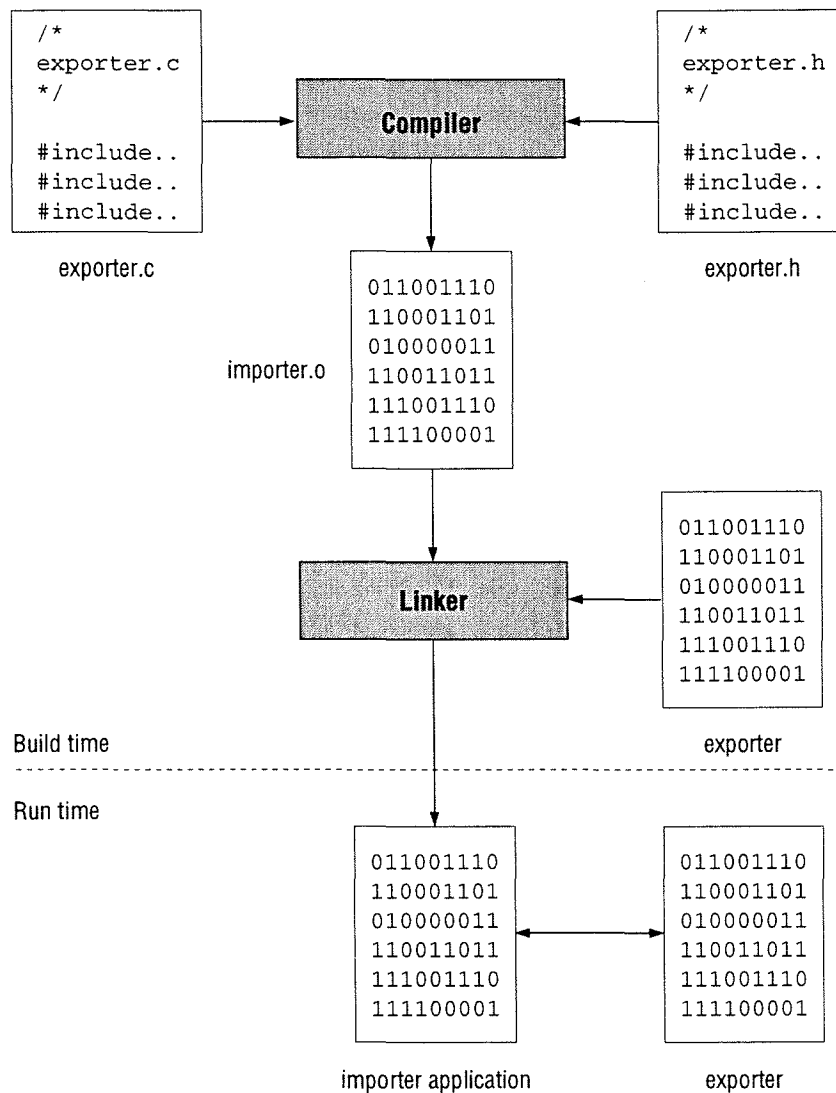


Figure 20-2 Building an importing module

Building an Exporting Module

Because the IMPORTER application is made up of two modules—a DLL and an application—it takes two build operations to build the complete application. To build the EXPORTER DLL, you must use four files: EXPORTER.C, EXPORTER.H, EXPORTER.X, and EXPORTER.MAKE. In this example, the EXPORTER.MAKE file is a makefile generated automatically by MPW.

Figure 20-1 on page 293 illustrates the process that was used to build the EXPORTER DLL. As the diagram shows, the compiler uses the EXPORTER.C and EXPORTER.H to generate a non-executable object file named EXPORTER.O. Then the linker uses the EXPORTER.O file and the EXPORTER.X definition file to generate the exporter module, which is named EXPORTER.

Building and Executing an Importing Module

To use the functions that are implemented in a DLL, you must build an application that imports those functions. The application that uses the functions implemented in the EXPORTER DLL shown in Figure 20-1 on page 293 is the IMPORTER application. Figure 20-2 on page 294 shows the procedure that is used to build and execute the IMPORTER program.

As Figure 20-2 illustrates, the procedure for building an importing module (in this case, an application) is very similar to the procedure for building a DLL. First, the M2 compiler uses the IMPORTER.C file and the EXPORTER.C header file to build a non-executable object file named IMPORTER.O. Then the linker uses both the IMPORTER.O file and the EXPORTER DLL (which was built using the procedure shown in Figure 20-1) to generate the IMPORTER application.

When the IMPORTER application has been built, it can import functions dynamically from the EXPORTER DLL. That's because the linker uses the EXPORTER file to provide the final executable version of the IMPORTER application with all the information it needs to import symbols from the EXPORTER DLL.

Linking an Application with a DLL

When you have created these three kinds of files, you can compile them and then link them by using the `-x` option on your linker command line. For example, the following LINK3DO command links the EXPORTER.O object file with the LINKEXEC.X definition file, as shown in Figure 20-1 on page 293:

```
link3do -r export.o -o -x linkexec.x exporter
```

When you execute this command line, the linker uses the EXPORTER.O file as its input. The linker also reads the symbol definitions in from the LINKEXEC.X definition file, and generates a module (in this case, a DLL module) named EXPORTER.DLL.

When the linker builds the binary executable file named EXPORTER DLL, it places in the EXPORTER file (in binary format, of course) a unique ID number that identifies the module. Numbers that identify all symbols exported by the EXPORTER DLL are also placed in the EXPORTER file. (For more information on ordinal numbers, see "Example of a Definition File" on page 291.)

In the preceding example:

- ◆ `-r` specifies that the linked file is to be relocatable
- ◆ `-x` identifies the definitions file for this object
- ◆ `exporter.o` is the compiled (but non-executable) object file that is compiled from the `EXPORTER.C` source file.
- ◆ `EXPORTER` is the name of the DLL that is generated by the build operation shown in Figure 20-1.

For more information about using the M2 linker, see your M2 linker documentation. For an example that shows how these three files can be used to implement a DLL, see "Example: Creating and Using a DLL" on page 300.

Functions for Handling Modules

The M2 Kernel provides a number of function calls for available for creating, loading, unloading, and manipulating modules. This section examines the module-related functions that the Kernel provides.

Using Items as Arguments and Return Values

To understand how the Kernel's module-related functions work, it is helpful to know that M2 treats modules as items. As explained in Chapter 6, "Managing Items", items are system-maintained handles for shared resources. Every item contains a globally unique identification number and a pointer to its associated resource. Items can refer to any one of many different system components such as data structures, I/O devices, folios, tasks, and so on. For more detailed information about items, see Chapter 6, "Managing Items".

Some module-related calls, such as `OpenModule()` and `ImportByAddress()`, use items as return values. Other calls, such as `CloseModule()` and `ImportByAddress()`, expect items to be passed to them as arguments.

The Kernel provides a special item, named *MyModule*, that always refers to the module currently being executed. You should always let the system control the value of *MyModule*; if you attempt to redefine the value of *MyModule* yourself, the result is undefined and could be a program crash.

Opening, Loading, and Executing Modules

The `OpenModule()` function loads a DLL module from disk and prepares it for use.

```
Item OpenModule(const char *path, OpenModuleTypes type, const TagArg *tags)
```

When you load a module by calling `OpenModule()`, you must specify the module's pathname and provide a *type* argument that specifies how memory for the module should be allocated. If the value of the *type* argument is

`OPENMODULE_FOR_THREAD`, memory is allocated within the current task's pages. If the value of *type* is `OPENMODULE_FOR_TASK`, memory is allocated from new pages.

Currently, the *tags* argument should be a `NULL` value.

When you have opened a DLL by calling `OpenModule()`, your application can spawn it as a thread or execute it as a subroutine. To execute the DLL, you can call `ExecuteModule()`, `CreateModuleThread()`, or `CreateTask()`. When you no longer need the DLL, you can remove it from memory by calling `CloseModule()`.

Closing a Module

The `CloseModule()` function closes a DLL module that has been opened with a call to `OpenModule()`.

```
Err CloseModule(Item module)
```

Once `CloseModule()` is called to close a DLL module, the module can no longer be used, and the system is free to unload it from memory. A successful call to `CloseModule()` returns a value of zero or greater. An error returns a negative error message.

Executing a Module

When you have loaded a module by calling `OpenModule()`, you can call `ExecuteModule()` to execute functions that are implemented in that module. As explained under the preceding headings, code implemented in such a module can run as a subroutine of the current task or thread.

```
int32 ExecuteModule(Item module, uint32 argc, char **argv)
```

The *argc* and *argv* parameters that you pass to `ExecuteModule()` are not used by the `ExecuteModule()` function. They are simply passed through to the loaded code. However, their meanings must be consistent with the requirements of the caller of the `ExecuteModule()` function, as well as consistent with the requirements of the executing application.

The return value of the `ExecuteModule()` function is the value returned by the `main()` function of the executing application.

Importing and Exporting Modules

The M2 Kernel provides several calls that you can use to load modules by name or by address. Calls that load modules begin with the word *Import*. Calls that export modules begin with the word *Unimport*.

When you are finished with a module that you have loaded by calling an `Importxxx` function, you should balance the `Importxxx` call that you used to import the function with a corresponding `Unimportxxx` call. This precaution is necessary because M2 keeps track of imported and unimported modules by

incrementing and decrementing counters. Because exported symbols can be accessed by multiple modules, failure to balance `Importxxx` and `Unimportxxx` calls can cause modules to be unloaded from memory while they are still being accessed by other modules.

Loading a Module by Specifying Its Name

You can call the `ImportByName()` function to load an exporting module by specifying its name.

```
Item ImportByName(Item module, const char *name)
```

The *module* variable is the number of the currently executing module, so you can pass the predefined *MyModule* item number in this parameter. The *name* variable is the name of the module being imported.

The return value is either the item module that has been loaded or an error code.

Loading a Module by Specifying Its Address

Call `ImportByAddress()` to load an exporting module by specifying a symbol exported by the module. The `ImportByAddress()` function can come in handy when you want to load a module and don't know the module's name.

```
Item ImportByAddress(Item module, void *address)
```

The *module* variable is the predefined *MyModule* item. The *address* variable is the address of the module being imported.

A successful call to `ImportByAddress()` returns the item number of the loaded module. An unsuccessful call returns an error code.

Unloading a Module by Specifying Its Name

You can use the `UnimportByName()` function to unload a module by specifying its name.

```
Err UnimportByName(Item module, const char *name)
```

The *module* variable is the predefined *MyModule* item. The *name* variable is the name of the module being unloaded.

A successful call to `UnimportByName()` unloads the named module. An unsuccessful call returns an error code.

Unloading a Module by Specifying Its Address

Use the `UnimportByAddress()` function to unload an exporting module by specifying a symbol exported by the module.

```
Err UnimportByAddress(Item module, const void *address)
```

The *module* variable is the predefined *MyModule* item. The *address* variable is the address of the module being imported.

A successful call to `UnimportByAddress()` unloads an exporting module, based on the address of an exported symbol. An unsuccessful call returns an error code.

Removing Symbols from the List of Available Symbols

The M2 Kernel provides two functions — `ExpungeByAddress()` and `ExpungeBySymbol()` — that you can use to remove exported symbols from your application's list of available symbols so they can no longer be used by importing modules.

Expunging a Symbol by Specifying Its Name

You can call `ExpungeBySymbol()` function to remove a symbol from an application's list of exports by specifying its name.

```
Err ExpungeBySymbol(Item module, int32 symbolNumber)
```

The *module* variable is the module from which the symbol is exported. The *name* variable is the name of the module being expunged.

A successful call to `ExpungeBySymbol()` removes the specified symbol from your application's exports table. An unsuccessful call returns an error code.

Expunging a Symbol by Specifying Its Address

Call `ExpungeByAddress()` when you need to remove a symbol from an application's list of exports by specifying its address.

```
Err ExpungeByAddress(Item module, const void *address)
```

The *module* variable is the module from which the symbol is exported. The *address* variable is the address of the module being expunged.

A successful call to `ExpungeByAddress()` removes a specified symbol from the exports table of a loaded module. An unsuccessful call returns an error code.

Looking Up the Address of a Symbol

The `LookupSymbol()` function returns the address of a loaded symbol by iterating over all symbols in the specified module.

```
void *LookupSymbol(Item module, int32 symbolNumber)
```


The *module* variable is the module item to be searched. The *number* variable is the item number of the module being looked up.

A successful call to `LookupSymbol()` returns the address of a exported symbol. An error returns an error code.

Example: Creating and Using a DLL

This section presents a pair of example programs that shows how you can create and use DLLs in M2 application. The example, named IMPORTER, is divided into two parts. The first part shows how to build and execute a program named IMPORTER.C.

The IMPORTER section of the IMPORTER program uses the EXPORTER portion to construct an example game. The EXPORTER portion of the program is the section that implements and exports a DLL.

The EXPORTER program has two entry points: one named `LoadGame` and one named `SaveGame`. These two entry points simulate the kinds of functions that a game would use only occasionally. Functions that are designed to be used only from time to time don't have to be memory-resident throughout a game, so they are good candidates for implementation as DLLs. That's because DLLs can be loaded into memory when they are needed and unloaded when they are not. The IMPORTER program show how DLLs can be unloaded from memory when they are not needed so the memory they occupy can be used for other purposes.

Building a DLL Step by Step

To build a symbol-exporting program such as the EXPORTER DLL, these are the steps to follow:

1. Create the source files that implement the DLL. (In the IMPORTER example, the DLL source file that is created is the EXPORTER.C file presented in Example 20-3 on page 302).
2. Create a linker definition file. The linker definition file for the EXPORTER.C file is the EXPORTER.X file presented in Example 20-6 on page 306. A linker definition file lists the symbols that are exported by a DLL module.
3. Create a makefile for your DLL module. In MPW, you can do this by invoking the `MPW CreateM2Make()` function. The makefile for the EXPORTER module, which was created using `CreateM2Make()`, is shown in Example 20-6 on page 306. In the EXPORTER makefile, you may notice that an `"a -xexporter.x"` directive has been added to the link line, so that the executable that is being built will export its symbols properly.
4. Execute the MPW `"BuildProgram exporter"` command to build your DLL.

Importing a DLL Step by Step

To build the portion of a program that imports a DLL, these are the steps to follow:

1. Create a file containing source code that imports a DLL. In the IMPORTER example, the source file that imports a DLL is the IMPORTER.C file shown in Example 20-5 on page 304.
2. Create a makefile for the portion of your program that imports a DLL. In MPW, you can do this by invoking the MPW CreateM2Make function. In the IMPORTER.MAKE file, shown in Example 20-8 on page 308, you may notice that the EXPORTER module has been added to the MODULES equate. This addition allows the executable to bind to the symbols provided by exporter.
3. Execute the MPW "BuildProgram exporter" command to build this portion of your program.

Executing the IMPORTER Program

To execute the IMPORTER program, follow these steps:

1. Copy the IMPORTER and EXPORTER executables into the M2 remote folder.
2. Execute the IMPORTER example.

Output of the IMPORTER Program

The output of the IMPORTER program is as follows:

```
-Normally a game would be asking the user what they want to do.  
-Since this is just a demo, we just provide fake user choices.
```

```
We just 'loaded' a savegame.
```

```
-Normally a game would be asking the user what they want to do.  
-Since this is just a demo, we just provide fake user choices.
```

```
Welcome to our demonstration game.  If this had been an actual  
game, we would be playing a level here.
```

```
-Normally a game would be asking the user what they want to do.  
-Since this is just a demo, we just provide fake user choices.
```

```
We just 'saved' the game for you
```

Example 20-3 *The EXPORTER.C File*

```
/*
    exporter.c
*/

#include "exporter.h"
#include <stdlib.h>
#include <string.h>

Err    LoadGame(struct GameInfo *gi)
/*
    In an actual game, this function would prompt the user to load
    a saved game.  The caller provides a pointer to a structure which
    will be initialized based on the user's input.

    If this function returns a negative value, then an error occurred
    or the user cancelled the load.  In either case, the GameInfo structure
    is left in an undefined state
*/
{
    /* In an actual game, this function would construct a graphical interface
       to give the user a choice on which savegame to load.  Here we just
       do some fake stuff.
    */

    printf("We just 'loaded' a savegame.\n\n");
    strcpy(gi->playerName, "Kevin 3DO");
    gi->score = 602;
    gi->level = 42;

    return 0;
}

Err    SaveGame(const struct GameInfo *gi)
/*
    This function saves the specified game state.  If an error value is
    returned then no save has actually taken place.
*/
{
    /* We also provide a fake savegame function*/

    printf("We just 'saved' the game for %s\n\n", gi->playerName);
}
```

```
    return 0;
}
```

Example 20-4 *EXPORTER.H: The EXPORTER program's header file*

```
/*
    exporter.h

    This file contains the prototypes for the functions which are exported by
    the example export module
*/

#include <kernel/types.h>

struct GameInfo
{
    char playerName[32];
    uint32 score;
    uint32 level;
};

Err LoadGame(struct GameInfo *gi);
/*
    In an actual game, this function would prompt the user to load
    a saved game. The caller provides a pointer to a structure which
    will be initialized based on the user's input.

    If this function returns a negative value, then an error occurred
    or the user cancelled the load. In either case, the GameInfo structure
    is left in an undefined state
*/

Err SaveGame(const struct GameInfo *gi);
/*
    This function saves the specified game state. If an error value is
    returned then no save has actually taken place.
*/
```

Example 20-5 *The IMPORTER.C File*

```
/*
    importer.c

    This module pretends to be the main module of a real game.
*/

#include <stdlib.h>
#include <stdio.h>
#include <kernel/loader3do.h>
#include "exporter.h"

typedef enum
{
    loadGame,
    playLevel,
    saveGame,
    quitGame
}
UserChoice;

structGameInfo ourGame;

void playLevel(void)
{
    printf("Welcome to our demonstration game.  If this had been an actual game,\n"
           "we would be playing a level here.\n\n");
}

UserChoice chooseLevel(void)
{
    static UserChoice choice = loadGame;
    UserChoice oldChoice;

    printf("-Normally a game would be asking the user what they want to do.\n"
           "-Since this is just a demo, we just provide fake user choices.\n\n");

    oldChoice = choice;
    choice++;

    /* We will claim we want to load, then play, then save, then quit*/

    return oldChoice;
}
```

```
}

voidDLLLoadGame(void)
{
    /* Since loading and saving games is done rarely, we don't keep the code in
       memory for this. We use a DLL to load and unload it as needed.
    */

    /* Real code should check for errors... ahem.*/
    LoaderImportByAddress(MyModule, LoadGame);

    /* At this point the code is loadedand can be called*/
    LoadGame(&ourGame);

    /* Now unload the DLL*/
    LoaderUnimportByAddress(MyModule, LoadGame);
}

voidDLLSaveGame(void)
{
    /* Since loading and saving games is done rarely, we don't keep the code in
       memory for this. We use a DLL to load and unload it as needed.
    */

    /* Real code should check for errors... ahem.*/
    LoaderImportByAddress(MyModule, SaveGame);

    /* At this point the code is loadedand can be called*/
    SaveGame(&ourGame);

    /* Now unload the DLL*/
    LoaderUnimportByAddress(MyModule, SaveGame);
}

voidmain(void)
{
    UserChoice choice;

    do
    {
        choice = ChooseLevel();
    }
```

```
switch(choice)
{
    case loadGame:DLLLoadGame();
        break;

    case saveGame:DLLSaveGame();
        break;

    case playLevel:PlayLevel();
        break;

    default:exit(0);
        break;
}
}
while(1);
}
```

Example 20-6 *The EXPORTER.X File*

```
!
! exporter - the save/load API
!

MAGIC16384
EXPORTS0=SaveGame
EXPORTS 1=LoadGame
```

Example 20-7 *The EXPORTER.MAKE File (Generated by MPW)*

```
# This Makefile was generated automatically by CreateM2Make

OBJECTS = 0
        :objects:exporter.c.o

# Choose one of the following:
# DEBUG_OPTIONS = -g # For best source-level debugging
# DEBUG_OPTIONS = -XO -Xunroll=1 -Xtest-at-bottom -Xinline=5
```

```

# variables for compiler tools
defines= -D__3DO__ -DOS_3DO=2 -DNUPUPSIM

dccopts= 		-
		-c -Xstring-align=1 -Ximport -Xstrict-ansi -Xunsigned-char 	-
		{DEBUG_OPTIONS} 	-
		-Xforce-prototypes -Xlint=0x10 -Xtrace-table=0

appname= exporter

LIBS = -lspmath -lfile -leventbroker -lc
MODULEDIR= "{3doremote}"System.m2:Modules:
MODULES=

{appname} ff {appname}.make {OBJECTS}
	 link3do -r -D -Htime=now -Hsubsys=1 -Hname={appname} -Htype=5 -Hstack=32768 	-
		-xexporter.x 	-
		-o {appname} 	-
		{OBJECTS} 	-
		-L"{3dolibs}{m2librelease}" 	-
		{MODULES} 		# 	-
		{LIBS} 		# 	-
# generate extra map info and redirect from dev:stdout to a map file
	 > {appname}.map
# Set creator and type so icon will appear
	 setfile {appname} -c '3DOD' -t 'PROJ'
	 duplicate {appname} -y {3doremote}# copy the goodies to /remote
#.. Make the .spt script, so that 3DODDebug finds the source files.
#.. 3DODDebug executes the script <appname>.spt when you debug <appname>.
#.. This default assumes all your source files are in the current directory
#.. at build time.
#.. Clear 3DODDebug's current list.
	 echo "setsourcedir" > "{3doremote}{appname}.spt"

# Tell 3DODDebug to look for source files in the current directory.
	 echo "setsourcedir 	-`directory`	-"" >> "{3doremote}{appname}.spt"
#.. <<If you have more than one source directory, insert more lines here.>>
# Clear 3DODDebug's current list of symbol files.
	 echo "setdatadir" >> "{3doremote}{appname}.spt"
	 echo "setdatadir 	-"{3doremote}	-"" >> "{3doremote}{appname}.spt"
# (Because currently the symbols are in the executable file).

# files in the objects sub-dir are dependent upon sources in the current dir
:objects: f :

# .c.o files are dependent only upon the .c source (This is not very complete
# but good enough for now)

```



```
.c.o f .c
    dcc {defines} {dccopts} {depDir}{default}.c -o {targDir}{default}.c.o
```

Example 20-8 *The IMPORTER.MAKE File (Generated by MPW)*

```
# This Makefile was generated automatically by CreateM2Make

OBJECTS = 0
    :objects:importer.c.o

# Choose one of the following:
DEBUG_OPTIONS = -g # For best source-level debugging
# DEBUG_OPTIONS = -XO -Xunroll=1 -Xtest-at-bottom -Xinline=5

# variables for compiler tools
defines= -D__3DO__ -DOS_3DO=2 -DNUPUPSIM

dccopts= 0
    -c -Xstring-align=1 -Ximport -Xstrict-ansi -Xunsigned-char 0
    {DEBUG_OPTIONS} 0
    -Xforce-prototypes -Xlint=0x10 -Xtrace-table=0

appname= importer

LIBS = -lspmath -lfile -leventbroker -lc
MODULEDIR= "{3doremate}"System.m2:Modules:
MODULES= exporter

{appname} ff {appname}.make {OBJECTS}
    link3do -r -D -Htime=now -Hsubsys=1 -Hname={appname} -Htype=5 -Hstack=32768 0
    -o {appname} 0
    {OBJECTS} 0
    -L"{3dolibs}{m2librelease}" 0
    {MODULES} # 0
    {LIBS} # 0
    > {appname}.map# generate extra map info and redirect from dev:stdout to a
map file
    setfile {appname} -c '3DOD' -t 'PROJ'# Set creator and type so icon will appear
    duplicate {appname} -y {3doremate}# copy the goodies to /remote
    #.. Make the .spt script, so that 3DODDebug finds the source files.
    #.. 3DODDebug executes the script <appname>.spt when you debug <appname>.
```

```

#... This default assumes all your source files are in the current directory at
build time.
# Clear 3DODebug's current list.
echo "setsourcedir" > "{3doremote}{appname}.spt"
# Tell 3DODebug to look for source files in the current directory.
echo "setsourcedir 0"directory`0"" >> "{3doremote}{appname}.spt"
#... <<If you have more than one source directory, insert more lines here.>>
# Clear 3DODebug's current list of symbol files.echo "setdatadir" >>
"{3doremote}{appname}.spt"
# (Because currently the symbols are in the executable file).

echo "setdatadir 0"{3doremote}0"" >> "{3doremote}{appname}.spt"
# files in the objects sub-dir are dependent upon sources in the current dir
:objects: f :

# .c.o files are dependent only upon the .c source (This is not very complete but
good enough for now)
.c.o f .c
    dcc {defines} {dcopts} {depDir}{default}.c -o {targDir}{default}.c.o

```


The Debug Console Link Library

This chapter describes the debugging console link library, which speeds up debugging by sending debugging output to a View on the 3DO display itself, instead of to the 3DO debugger.

This chapter contains the following topics:

Topic	Page Number
About the Debug Console Link Library	311
Preparing For Console Output	312
Performing Console Output	312
Concluding Console Output	313

Note: For detailed descriptions of individual function calls, consult the 3DO M2 Portfolio Programmer's Reference.

About the Debug Console Link Library

Outputting text to the standard debugging terminal using `printf()` is fairly slow, involving a considerable amount of latency. As a result, it is often painful to output information within a real-time program, since the fact of calling `printf()` tends to radically change the speed of the running program and destroys any hope of maintaining real-time performance.

To help compensate for this poor performance, the debugging console package provides a set of functions to send debugging output to a View on the 3DO display itself, instead of to the 3DO debugger. This avoids all of the latency issues involved with communicating with the debugger and results in much less impact on the performance of the client program.

Preparing For Console Output

Before you can perform output using the debugging console, you must first call the `CreateDebugConsole()` or `CreateDebugConsoleVA()` functions:

```
Err CreateDebugConsole(const TagArg *tags);  
Err CreateDebugConsoleVA(uint32 tag, ...);
```

You can call `CreateDebugConsole()` with a `NULL` parameter. This operation results in a half-screen 640-by-240 pixel View being opened on the 3DO display. You can also supply tags to either function to control attributes of the View being created.

The `DEBUGCONSOLE_TAG_HEIGHT` tag lets you specify how many pixels tall the view should be. It is sometimes useful to open a view which is less than full-screen in order to see underlying views showing part of the program being debugged.

The `DEBUGCONSOLE_TAG_TOP` tag lets you specify how many pixels from the top of the display the debugging view should be opened. The `DEBUGCONSOLE_TAG_TYPE` tag lets you specify the exact graphics view type of the debugging view. The `<graphics:view.h>` include file contains a list of the possible view types. The types control the resolution or the view.

Here's an example call to `CreateDebugConsoleVA()`:

```
err = CreateDebugConsoleVA(DEBUGCONSOLE_TAG_HEIGHT, 200,  
                           DEBUGCONSOLE_TAG_TOP,    150,  
                           TAG_END);  
  
if (err >= 0)  
{  
    /* the debugging view is now opened and functional */  
}
```

The above code opens a view which is 200 pixels tall and starts 150 pixels from the top of the display.

Performing Console Output

Once a debugging view has been created, three simple operations can be performed to the view:

- ◆ Outputting text

- ◆ Changing the text cursor position
- ◆ Clearing the view

To output text, you use the `DebugConsolePrintf()` call. The function works exactly like the standard C `printf()` routine, except that the output is sent to the debugging view instead of to the standard debugger terminal window.

```
void DebugConsolePrintf(const char *text, ...);
```

As text is printed, it will automatically begin to scroll as the bottom of the view is reached. Once text scrolls off the top of the view, there is currently no way to recover it.

You can erase everything in the debugging view and reset the position to the top of the display by using `DebugConsoleClear()`:

```
void DebugConsoleClear(void);
```

You can also explicitly position the cursor within the view by using `DebugConsoleMove()`:

```
void DebugConsoleMove(uint32 x, uint32 y);
```

The x and y coordinates specify a location relative to the top left of the display where the next burst of text written out will start.

Concluding Console Output

When you are done with the debugging console, you can close it down and free any resources it was using by calling `DeleteDebugConsole()`:

```
void DeleteDebugConsole(void);
```


The Script Folio

This chapter documents the Script folio functions that enable you to invoke shell commands and scripts from a program.

This chapter contains the following topics:

Topic	Page Number
About the Script Folio	315
Executing Commands	316
Preserving State Across Multiple Executions	316

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

About the Script Folio

The Script folio acts as an extension of the Portfolio shell. The Script folio contains shell commands that are normally used only by developers during development of a game but are not normally needed when customers actually run the game. To conserve memory, the portion of the shell loaded into RAM during normal game operation is kept as small as possible. The rest of the shell commands are in the Script folio, which is only loaded into RAM when needed.

Along with shell commands, the Script folio provides C-callable functions that let you execute a shell command-line from within your program as if it had been typed into the shell in response to the shell prompt. Note that the use of these functions at run time will cause the Script folio to be loaded into RAM. It is these functions that are described in this chapter.

Core shell commands and Script folio shell commands are documented in another manual entitled *3DO M2 Debugger Programmer's Guide*.

Executing Commands

You can easily execute any command line from within your programs, just as if it had been typed into the shell. This means you can start programs, scripts, and built-in commands. To execute a command-line, you use the `ExecuteCmdLine()` or `ExecuteCmdLineVA()` functions:

```
Err ExecuteCmdLine(const char *cmdLine, int32 *pStatus, const TagArg
                  *tags);

Err ExecuteCmdLineVA(const char *cmdLine, int32 *pStatus, uint32 tag,
                    ...);
```

The `cmdLine` argument is the command-line to execute, as if it were typed in at a shell prompt. This may be the name of a program to load and run, the name of a built-in command such as `ShowTask`, or the name of an ASCII script file. The `pStatus` argument points to a variable where the execution status of the program being run will be stored. This is the value returned by a program's `main()` function. Finally, the `tags` argument lets you supply additional optional arguments to the function.

The `SCRIPT_TAG_CONTEXT` tag lets you supply a pointer to a `ScriptContext` structure. See the next section for information on the `ScriptContext`.

The `SCRIPT_TAG_BACKGROUND_MODE` lets you set whether the command line should be executed in foreground or background mode. When in background mode, programs being started will run asynchronously and this function returns while the program is still running.

If you launch a script in background mode, the individual programs within the script will be launched asynchronously and this function will return once all programs are running. Within a script, the foreground and background mode can be changed by using the `SetFg` and `SetBg` commands.

Preserving State Across Multiple Executions

When a script is running, it is possible for the script to change its environment. For example, a script can change its background mode setting. If you call `ExecuteCmdLine()` without supplying a `ScriptContext` structure, any changes made to the environment of a script are lost as soon as `ExecuteCmdLine()` returns. By creating a `ScriptContext` structure, the environment can be preserved and reused within subsequent calls to `ExecuteCmdLine()`. This is how the shell manages to keep its state even though it makes a different call to `ExecuteCmdLine()` every time the user enters a string at the shell prompt.

You call `CreateScriptContext()` to create a `ScriptContext` structure:

```
Err CreateScriptContext(ScriptContext **sc, const TagArg *tags);  
Err CreateScriptContextVA(ScriptContext **sc, uint32 tag, ...);
```

The `sc` argument is a pointer to a variable that receives the `ScriptContext` pointer. The `tags` argument lets you supply additional optional parameters. The only tag that is currently supported is the `SCRIPT_TAG_BACKGROUND_MODE` tag that lets you adjust the initial setting of the background execution mode of the `ScriptContext`.

Once a script context is created, you can use it in any number of calls to `ExecuteCmdLine()`. Any changes to the execution environment are recorded within the `ScriptContext` and are therefore preserved across the different calls.

When you're done with a `ScriptContext` structure, you call `DeleteScriptContext()` to dispose of it:

```
Err DeleteScriptContext(ScriptContext *sc);
```


Using Lumberjack, the Event Logger

This chapter describes Lumberjack, which is a collection of functions in the kernel that lets you log events to aid during debugging. The log information includes the type of event, the time at which it occurred, the task that triggered the event, and relevant parameters for that type event. This chapter explains how to use Lumberjack to log and parse events.

This chapter contains the following topics:

Topic	Page Number
Overview	319
Collecting Information With Lumberjack	320
Getting Information From Lumberjack	321
Example: Using Lumberjack	323

Note: For detailed descriptions of individual function calls, consult the *3DO M2 Portfolio Programmer's Reference*.

Overview

Lumberjack lets you log a large selection of kernel events, including task switches, interrupts, message passing, semaphore locking, memory allocations, item manipulation, I/O operations, and more. In addition, you can also add your own custom events to the logs.

The log information is recorded into memory buffers. It can be extracted from those buffers when convenient and analyzed or dumped to the debugging terminal. You can control which events are logged, and when logging occurs.

Collecting Information With Lumberjack

This section discusses all aspects of collecting information with Lumberjack. The process consists of

- ◆ **Creating Lumberjack**, which allocates the buffers.
- ◆ **Starting and Stopping Event Logging**.
- ◆ **Logging Custom Events** (optional)
- ◆ **Deleting Lumberjack**

Creating Lumberjack

To use Lumberjack, first call `CreateLumberjack()` to allocate and initialize the needed data structures:

```
Err CreateLumberjack(const TagArg *tags)
```

The `tags` argument is reserved for the future and must currently always be `NULL`. The function returns ≥ 0 if Lumberjack was created, or a negative error code for failure.

When you call this function, a list of buffers is allocated that currently total up to a little over 768 KB of memory. These buffers are used by Lumberjack to store the log entries. To start logging information in these buffers, call the `ControlLumberjack()` function discussed in the next section.

Starting and Stopping Event Logging

To start and stop event logging, use the `ControlLumberjack()` function:

```
Err ControlLumberjack(uint32 eventsLogged)
```

You pass this function a mask of events to log. The mask is constructed by using the many `LOGF_CONTROL_XXX` constants defined in `<kernel/lumberjack.h>`. By combining these constants, you can cause many different kinds of events to be logged.

Only the events specified in the argument are logged. You can therefore turn off logging for different event classes by clearing the appropriate bits within the function argument. If you supply 0 for the argument, all logging is stopped.

The function returns ≥ 0 if it succeeded, or a negative error code for failure.

Logging Custom Events

Lumberjack can log a large collection of kernel events automatically. It also lets you log your own events. This can be very useful when trying to understand how a program behaves. To log your own events, call the `LogEvent()` function:

```
Err LogEvent(const char *eventDescription)
```

`LogEvent()` takes one parameter: a descriptive string that is recorded into the logs. The log entry contains the time at which the event was logged, as well as the name and item of the task that logged the event.

Deleting Lumberjack

When you are done logging information, you can delete Lumberjack by calling the `DeleteLumberjack()` function:

```
Err DeleteLumberjack(void)
```

This function frees all the resources used by Lumberjack and discards any events currently in the event logs.

Getting Information From Lumberjack

Getting information from Lumberjack often consists of two steps:

- ◆ Getting and Releasing Buffers
- ◆ Parsing Lumberjack Buffers

Getting and Releasing Buffers

At any time during the logging process, you can ask Lumberjack to supply you with an event buffer. Once you have a buffer, you can parse its contents in order to determine the sequence of events. To get a buffer, call the function `ObtainLumberjackBuffer()`:

```
LumberjackBuffer *ObtainLumberjackBuffer(void)
```

The function returns a pointer to a logging buffer which currently contains logged events. Once you have obtained such a buffer, you can parse its contents to extract useful information, or just call `DumpLumberjackBuffer()` to display the buffer contents to the debugging terminal in a standard format.

When you're done with a log buffer, you should return it to Lumberjack using the `ReleaseLumberjackBuffer()` function.

```
void ReleaseLumberjackBuffer(LumberjackBuffer *lb)
```

Releasing a buffer tells Lumberjack that you are done with it and that the buffer can be reused to store more log entries. If the buffer is not returned, Lumberjack may become unable to log events because it has no buffer space left.

Before you start to obtain buffers, it is a good idea to first disable event logging using `ControlLumberjack()`. Otherwise, new events will continuously be logged while you're dumping them or processing them, and you'll never see the end of it.

Parsing Lumberjack Buffers

A `LumberjackBuffer` is the data structure used by Lumberjack to log events. The buffer is organized to help Lumberjack be fast and efficient during the logging process. As a result it seems a bit clumsy when it comes time to parse its contents.

You can use the `DumpLumberjackBuffer()` function to display the contents of a `LumberjackBuffer` to the debugging terminal. The function displays all the information that was collected in an easy to read format.

`DumpLumberjackBuffer()` might not be what you need however. The amount of information that it displays can be quite large. Using `ControlLumberjack()`, you can easily control which information is logged, but even when logging the minimum needed, the amount of information generated can still be overwhelming. It is therefore often necessary to write custom routines to parse `LumberjackBuffer` structures. These routines can perform sophisticated analysis of the buffer contents to extract useful information.

The `Lumberjack.lb_BufferData` field points to the buffer's data, and the `Lumberjack.lb_BufferSize` field indicates how many bytes of data are in the buffer. The buffer data is organized as an array of events. Every event in this array is of variable size.

Every event logged starts with a `LogEventHeader` data structure. This structure identifies the type of event that is being described, the time at which the event occurred, the task that triggered the event, and the total size of the event structure. Following the header comes data that is specific to each event type. `<kernel/lumberjack.h>` contains data structure definitions for every type of event that can be generated.

The first event is stored at the beginning of the data area of a buffer. To find the address of the following event, you must add the value of the `leh_NextEvent` field to the address of the current event, and so on to progress through the buffer. The end of a buffer is marked by an event of type `LOG_TYPE_BUFFER_END`.

The `LumberjackBufferOffset` type is used throughout the event structures to denote string pointers. The values of these fields indicate offsets from the beginning of the buffer to where the string is located. By adding the address of the buffer with such a value, you will get a pointer to the string data itself. A `LumberjackBufferOffset` value of 0 indicates a NULL string.

If Lumberjack runs out of buffer space when trying to log an event, it will record the time at which the overflow condition started. The next time a buffer becomes available as a result of `ReleaseLumberjackBuffer()`, the fact that an overflow occurred is recorded as an event of type `LOG_TYPE_BUFFER_OVERFLOW`.

Example: Using Lumberjack

Example 23-1 Using Lumberjack

```
#define STR(offset) (offset ? ((char *)((uint32)lb->lb_BufferData
                                + (offset))) : "<null>")

void WalkLumberjackBuffer(const LumberjackBuffer *lb)
{
    LogEventHeader *leh;
    TimeVal tv;

    leh = (LogEventHeader *)lb->lb_BufferData;
    while (leh->leh_Type != LOG_TYPE_BUFFER_END)
    {
        ConvertTimerTicksToTimeVal(&leh->leh_TimeStamp, &tv);

        printf("Event Type: %d\n", leh->leh_Type);
        printf("Event Time: %d.%06d seconds\n",
               tv.tv_Seconds, tv.tv_Microseconds);

        if (leh->leh_TaskItem == -1)
        {
            printf("Current Task : <idle loop>\n");
        }
        else
        {
            printf("Current Task : %s (item %05x)\n",
                   STR(leh->leh_TaskName), leh->leh_TaskItem);
        }

        switch (leh->leh_Type)
        {
            /* do something with this event, based on its type */
        }

        leh = (LogEventHeader *)((uint32)leh + leh->leh_NextEvent);
    }
}
```


Index

A

AbortIO() PPG-113
AddHead() PPG-40
adding a node to a list PPG-40
AddTail() PPG-39, PPG-40, PPG-41
AddTimerTicks() PPG-123
AddTimes() PPG-118, PPG-122
AIFC FORM PPG-239
alignment
 specifying PPG-53
AllocateContextInfo() PPG-264, PPG-265
allocating a memory block PPG-53
allocating memory PPG-5
AllocContextInfo() PPG-253
AllocMem() PPG-5, PPG-52, PPG-53, PPG-59, PPG-61
AllocMemAligned() PPG-54
AllocMemBlocks() PPG-5, PPG-58
AllocMemFromLists() PPG-5
AllocMemPages() PPG-58
AllocSignal() PPG-13, PPG-26, PPG-84
analog joysticks PPG-199
anchor PPG-38
anchored list PPG-38
AppendPath() PPG-169
applications
 multi-module PPG-289
ASSERT() macros PPG-56
AtomicClearBits() PPG-65
AtomicSetBits() PPG-65
AttachContextInfo() PPG-264, PPG-266
audio controller
 subcodes PPG-213

AUDIO_TEMPLATE_NODE PPG-69
AUDIONODE PPG-69
avatars PPG-139
 placement PPG-139

B

Batt folio PPG-171
battery-backed clock PPG-171
block-oriented write PPG-127
Boyer-Moore algorithm PPG-65
break signals
 sending PPG-135
busy-waiting PPG-4

C

CacheInfo structure PPG-62
 fields PPG-62
Caches PPG-62
capture masks PPG-176, PPG-184, PPG-188
Catapult PPG-139
ChangeDirectory() PPG-152
ChangeItemOwner() PPG-11
character strings
 working with PPG-224
CheckIO() PPG-110
CheckItem() PPG-74
child tasks PPG-4, PPG-19
ClearBitRange() PPG-64
ClearCurrentSignals() PPG-86
ClearRawFileError() PPG-149
CloseCompressionFolio() PPG-229
CloseDirectory() PPG-152

CloseItem() PPG-11, PPG-75
 CloseModule() PPG-297
 CloseRawFile() PPG-151
 CMD_BLOCKWRITE PPG-127
 CMD_STATUS PPG-126
 CMD_STREAMREAD PPG-132
 CMD_STREAMWRITE PPG-131
 communication among tasks PPG-13
 CompareTimerTicks() PPG-123
 CompareTimes() PPG-122
 compress data PPG-228
 Compress() PPG-231
 compression engine
 creating PPG-228
 deleting PPG-229
 feeding PPG-229
 Compression folio PPG-227
 compression folio
 callback function PPG-230
 closing PPG-229, PPG-230
 convenience calls PPG-231
 how it works PPG-228
 how to use PPG-228
 opening PPG-229
 configuration data block PPG-183
 configuration messages PPG-183, PPG-189
 ConfigurationRequest data structure PPG-188
 console
 debugging PPG-311, PPG-312
 console output
 performing PPG-312
 control pad PPG-102
 monitoring PPG-215
 control pad return data PPG-194
 control port PPG-102
 control port devices PPG-102
 controlling memory allocations PPG-230
 ControlLumberjack() PPG-320
 ControlMem() PPG-9, PPG-57, PPG-58
 ControlMemDebug() PPG-59
 ControlPadEventData data structure PPG-194
 convenience calls PPG-214
 ConvertGregorianToTimeVal() PPG-173
 ConvertTimerTicksToTimeVal() PPG-123
 ConvertTimeValToGregorian() PPG-173
 ConvertTimeValToTimerTicks() PPG-123
 copy object
 creating PPG-167
 deleting PPG-168
 CreateBufferedMsg() PPG-14
 CreateBufferedMsg() PPG-90, PPG-183
 CreateCompressor() PPG-228, PPG-230
 CreateCopyObj() PPG-165, PPG-167
 CreateCustomSignalIOReq() PPG-107
 CreateDebugConsole() PPG-312
 CreateDebugConsoleVA() PPG-312
 CreateDecompressor() PPG-228, PPG-229, PPG-230
 CreateDeviceStackList() PPG-105
 CreateDirectory() PPG-151
 CreateFileInDir() PPG-141
 CreateIFFParser() PPG-251, PPG-254, PPG-256
 CreateIOReq() PPG-73, PPG-104, PPG-107, PPG-110, PPG-116
 CREATEIOREQ_TAG_SIGNAL PPG-106
 CreateItem() PPG-10, PPG-11, PPG-20, PPG-34, PPG-69, PPG-70, PPG-73
 CreateItemVA() PPG-34, PPG-72
 CreateLumberjack() PPG-320
 CreateMemDebug() PPG-59
 CreateModuleThread() PPG-25
 CreateMsg PPG-14
 CreateMsg() PPG-14, PPG-73, PPG-89
 CreateMsgPort() PPG-15, PPG-73, PPG-88, PPG-183
 CreateScriptContext() PPG-317
 CreateSemaphore() PPG-73, PPG-78, PPG-79
 CreateSmallMsg() PPG-14, PPG-90
 CreateStorageReq() PPG-283, PPG-285
 CreateTask PPG-26
 CreateTask() PPG-21, PPG-25
 CREATETASK_TAG_STACKSIZE PPG-21
 CreateThread() PPG-26
 CreateThread() PPG-24, PPG-25, PPG-26, PPG-73
 CreateTimerIOReq() PPG-117, PPG-118
 creating a compression engine PPG-228
 creating a linked list PPG-39
 creating a semaphore PPG-78
 current language
 determining PPG-223
 custom devices PPG-102
 custom events PPG-185

D

data structure PPG-46

data structures

FileSystemStat PPG-125

IOReq PPG-112

Link PPG-49

ListAnchor PPG-49

ListenerList PPG-210

Locale PPG-218

NoteTracker PPG-39

NumericSpec PPG-221

PodData PPG-212

PodDescriptionList PPG-208

SetFocus PPG-211

TagArg PPG-70

VBlankTimeVal PPG-120

date

Gregorian PPG-171

date and time

Gregorian

converting to TimeVal PPG-173

reading PPG-172

setting PPG-172

validating PPG-174

TimeVal

converting to Gregorian PPG-173

Date folio PPG-172

DateSpec arrays PPG-223

debug console link library PPG-311

DEBUGCONSOLE_TAG_HEIGHT tag PPG-312

DEBUGCONSOLE_TAG_TOP tag PPG-312

DebugConsoleClear() PPG-313

DebugConsoleMove() PPG-313

DebugConsolePrintf() PPG-313

debugging console PPG-311, PPG-312

preparing for output PPG-312

Decompress() PPG-231

decompressing data PPG-228

decompression engine

creating PPG-229

deleting PPG-229

feeding PPG-229

default context node PPG-241

definition file PPG-288, PPG-290

DeleteCompressor() PPG-228, PPG-229

DeleteCopyObj() PPG-165, PPG-168

DeleteDebugConsole() PPG-313

DeleteDecompressor() PPG-229

DeleteDirectory() PPG-151

DeleteFile() PPG-141

DeleteFileInDir() PPG-141

DeleteIFFParser() PPG-254, PPG-255, PPG-257

DeleteIOReq() PPG-114

DeleteItem() PPG-12, PPG-21, PPG-75

DeleteLumberjack() PPG-321

DeleteMemDebug() PPG-59

DeleteMemDebug() PPG-59

DeleteScriptContext() PPG-317

DeleteSemaphore() PPG-79, PPG-81

DeleteStorageReq() PPG-283, PPG-286

DeleteTree() PPG-168

deleting a semaphore PPG-80

deleting the compression engine PPG-229

device

opening PPG-105

device drivers PPG-102, PPG-104, PPG-115

device item PPG-137

device item number PPG-106

device stack PPG-105

device stacks PPG-104

devices PPG-101

control port PPG-102

daisy chain PPG-103

generic classes PPG-212

software PPG-103

devices in 3DO system PPG-115

directories PPG-138

renaming PPG-141

directory

changing PPG-152

creating PPG-151

deleting PPG-151

finding PPG-152

getting information PPG-130

opening and closing PPG-151

reading PPG-153

directory entry PPG-153

Directory functions PPG-138

directory functions PPG-151

Directory structure PPG-152

directory tree

copying PPG-165

deleting PPG-168

DismountFileSystem() PPG-153

DisplayStorageReq() PPG-283, PPG-285

DLL symbols PPG-289

DLLS PPG-287, PPG-309

DLLs

- advantages of using PPG-288
- benefits of using PPG-288
- creating and using PPG-288
- linking PPG-295

DLLs, see also dynamic-link library PPG-287

DoIO() PPG-111

DRAM PPG-52

DumpBitRange() PPG-66

DumpLumberjackBuffer() PPG-322

DumpMemDebug() PPG-59, PPG-60

DumpNode() PPG-47

DumpNode() PPG-48

DumpTagList() PPG-36

dynamic loading PPG-288, PPG-289

dynamic-link libraries PPG-287, PPG-309

E

EA IFF 85 Standard PPG-237

EB_IssuePodCmdReply PPG-214

EB_MakeTableReply message PPG-200

entry point

DLL PPG-288

error codes PPG-2

event broker PPG-175, PPG-176

configuring PPG-178

connecting to PPG-178, PPG-183, PPG-214

convenience calls PPG-214

disconnecting PPG-206, PPG-207, PPG-216

features PPG-178

file system state data structure PPG-199

high-performance use PPG-199

reconfiguring PPG-206

reconfiguring connection PPG-206

reply to command PPG-214

trigger mechanism PPG-190

event broker listeners

getting list of PPG-210

event broker message types PPG-181

event broker messages PPG-178

accompanying data structures PPG-182

configuration PPG-183

configuration reply PPG-189

flavors PPG-178

event capture mask PPG-176

event data

reading PPG-194

event masks PPG-176

event message data blocks PPG-190

event messages

configuration PPG-189

reading PPG-191

retrieving and replying PPG-191

event monitoring PPG-182, PPG-190

event notifications PPG-190

event queues PPG-184, PPG-191

event trigger mask PPG-176

event types PPG-185

EventBrokerHeader data structure PPG-189,
PPG-191

EventFrame data structure PPG-191

events PPG-176

custom PPG-185

monitoring PPG-176

non-UI PPG-185

specifying and monitoring PPG-176

system PPG-185

UI PPG-185

waiting for PPG-134

ExecuteCmdLine() PPG-316, PPG-317

ExecuteCmdLineVA() PPG-316

ExecuteModule() PPG-297

exit point

DLL PPG-288

exporting module

building PPG-293, PPG-294

ExpungeByAddress() PPG-299

ExpungeBySymbol() PPG-299

F

FeedCompressor() PPG-228, PPG-229

FeedDecompressor() PPG-229

FeedDecompressorEngine() PPG-228

feeding the compression engine PPG-229

feeding the decompression engine PPG-229

file

allocating blocks for PPG-127

definition PPG-290

finding PPG-142

getting status PPG-126

marking end of PPG-128

reading blocks PPG-128

reading data from PPG-128

setting attributes PPG-142

writing blocks to PPG-127

file device PPG-115
 file device driver PPG-123
 file devices
 communicating with PPG-123
 File folio PPG-138
 file folio
 examples PPG-154
 File functions PPG-138
 file system PPG-137
 finding PPG-154
 getting status PPG-125
 minimizing PPG-153
 mounting and dismounting PPG-153
 file system interface PPG-137
 file type
 setting PPG-129
 file version
 setting PPG-129
 file virtual block size
 setting PPG-129
 FILECMD_ALLOCBLOCKS PPG-127
 FILECMD_READDIR PPG-130
 FILECMD_READENTRY PPG-130
 FileInfo structure PPG-150
 files PPG-138
 opening and closing PPG-141
 renaming PPG-141
 structure of PPG-138
 filesystem
 examples PPG-154
 getting status PPG-125
 FileSystemStat structure PPG-125
 FindAndOpenItem() PPG-75
 FindAndOpenItemVA() PPG-75
 FindClearBitRange() PPG-65
 FindCollection() PPG-253, PPG-258
 FindContextInfo() PPG-254, PPG-266
 FindFileAndIdentify() PPG-142
 FindFileAndOpen() PPG-142
 FindFinalComponent() PPG-169
 finding a semaphore PPG-80
 FindItem() PPG-11, PPG-73
 FindMsgPort() PPG-96, PPG-189
 FindNamedItem() PPG-11, PPG-15, PPG-73
 FindNamedNode() PPG-45, PPG-47, PPG-48
 FindPropChunk() PPG-261, PPG-262
 FindSemaphore() PPG-81
 FindSemaphore() macro PPG-79
 FindSetBitRange() PPG-65

FindTagArg() PPG-35
 FindTask() PPG-21
 FirstNode() PPG-43, PPG-44
 FlushDCache() PPG-64
 FlushDCacheAll() PPG-64
 focus holder PPG-211
 folio management PPG-2
 forking tasks PPG-5
 formatting currency PPG-226
 formatting dates PPG-224
 formatting numbers PPG-226
 free memory pages PPG-52
 free() PPG-9
 FreeContextInfo() PPG-267
 freeing a memory block PPG-54
 FreeMem() PPG-9, PPG-53, PPG-54, PPG-59,
 PPG-61
 free-signal mask PPG-14
 FreeSignal() PPG-14, PPG-86
 FSUtils Folio PPG-165

G

game data
 loading into memory PPG-278
 saving PPG-276
 GetCacheInfo() PPG-62
 GetCompressorWorkBufferSize() PPG-230
 GetControlPad() PPG-215
 GetCurrentContext() PPG-268
 GetCurrentSignals() PPG-86
 GetDCacheFlushCount() PPG-63
 GetDecompressorWorkBufferSize() PPG-230
 GetDirectory() PPG-152
 GetIFFOffset() PPG-270
 GetMemInfo() PPG-55
 GetMemTrackSize() PPG-55
 GetMouse() PPG-215
 GetMsg() PPG-15, PPG-92, PPG-191
 GetNodeCount() PPG-46
 GetNodePosFromTail() PPG-46
 GetPageSize() PPG-5, PPG-56
 GetParentContext() PPG-268
 GetPath() PPG-169
 GetRawFileInfo() PPG-149
 GetTag() PPG-35
 GetThisMsg() PPG-94
 getting filesystem status PPG-125

glasses controller
 subcodes PPG-213
GregorianDate structure PPG-172

H

hardware devices PPG-102
header files
 audio/audio.h PPG-69
 kernel/io.h PPG-107
 kernel/kernelnodes.h PPG-70
 kernel/types.h PPG-70

I

I/O PPG-2, PPG-101, PPG-105
 aborting PPG-113
 finishing PPG-114
 multiple operations PPG-113
 preparing for PPG-105
I/O operations
 cleaning up after PPG-131
I/O process PPG-105
I/O request PPG-104
I/O requests PPG-104
 creating PPG-106
 deleting PPG-114
 reading PPG-112
 return notification PPG-106
Icon folio PPG-271, PPG-272
icons
 ICON FORM chunks PPG-272
 loading into memory PPG-272
 saving to a file PPG-274
 unloading from memory PPG-274
 uses PPG-271
IFF PPG-276
 !(c) chunk PPG-249
 !App chunk PPG-249
 !Mod chunk PPG-249
 !Rem chunk PPG-249
 !URL chunk PPG-250
 !Ver chunk PPG-249
 CAT chunks PPG-247
 collection chunks PPG-251, PPG-253, PPG-257
 finding PPG-258
 registering PPG-258
 container chunk PPG-238
 container chunks PPG-245, PPG-246
 context info structures PPG-253
 context stack PPG-240
 ContextInfo structures
 allocating PPG-265
 attaching PPG-266
 deallocating PPG-267
 finding PPG-266
 inserting PPG-266
 removing PPG-267
 using PPG-264
 default context node PPG-264
 EAIFF 85 PPG-237
 entry and exit handlers PPG-263
 entry handlers PPG-252, PPG-263
 exit handlers PPG-264
 file format PPG-238
 3DO extension PPG-245
 description PPG-243
 FORM chunks PPG-246
 getting current context node PPG-268
 getting parent context node PPG-268
 getting seek position in stream PPG-270
 global and local chunk ids PPG-247, PPG-248
 goals PPG-239
 LIST chunks PPG-247
 local chunk PPG-238
 parse operation
 starting or continuing PPG-256
 parser structure
 creating PPG-256
 deleting PPG-256, PPG-257
 popping a context node PPG-269
 PROP chunks PPG-247, PPG-259
 property chunks
 finding PPG-262
 registering PPG-261
 scoping rules PPG-259
 vs. collection chunks PPG-261
 vs. PROP chunks PPG-259
 pushing a context node PPG-268
 reading a chunk PPG-269
 reading and parsing a FORM PPG-250
 seeking in chunk PPG-270
 stop chunks PPG-252
 registering PPG-262

-
- use of in 3DO M2 PPG-240, PPG-243
 - writing a chunk PPG-269
 - writing out a FORM PPG-254
 - IFF folio PPG-237
 - capabilities PPG-240
 - description of functions PPG-255
 - purpose PPG-240
 - IFF_PARSE_SCAN PPG-252
 - IFF_SIZE_UNKNOWN_32 PPG-255, PPG-269, PPG-270
 - IFF_SIZE_UNKNOWN_64 PPG-255, PPG-269, PPG-270
 - IFFParser structure PPG-251
 - IFFTypeID structure PPG-261
 - IMPORT_FLAG flag PPG-292
 - IMPORT_NOW flag PPG-291
 - IMPORT_ON_DEMAND flag PPG-292
 - ImportByAddress() PPG-298
 - ImportByName() PPG-298
 - importing module
 - building and executing PPG-295
 - InitEventUtility() PPG-214
 - initializing a linked list PPG-39
 - Input focus PPG-177
 - input focus PPG-177, PPG-210
 - InsertNodeAlpha(PPG-48
 - InsertNodeAlpha() PPG-41, PPG-47
 - InsertNodeFromHead() PPG-39, PPG-41, PPG-48
 - InsertNodeFromTail(PPG-39, PPG-48
 - InsertNodeFromTail() PPG-41
 - InstallEntryHandler() PPG-252
 - InstallExitHandler() PPG-264
 - international folio
 - character strings PPG-224
 - determining current language PPG-223
 - formatting dates PPG-224
 - formatting numbers PPG-226
 - internationalization PPG-217
 - intertask communication PPG-2, PPG-13
 - IO_QUICK PPG-111
 - ioi_Command PPG-119
 - IOInfo PPG-105
 - IOInfo data structure PPG-104, PPG-105, PPG-107
 - setting values PPG-109
 - IOInfo fields
 - setting PPG-116
 - IOInfo structure PPG-116
 - fields PPG-108
 - initializing PPG-107
 - IOInfo.ioi_CmdOptions PPG-116
 - IOInfo.ioi_Recv PPG-116
 - IOInfo.ioi_Send PPG-116
 - IOReq PPG-104, PPG-106
 - creating PPG-73
 - sending PPG-124
 - IOReq data structure PPG-105, PPG-106, PPG-112, PPG-116
 - IOReq structure PPG-105
 - IOReqs PPG-104
 - isBitClear() PPG-65
 - IsBitRangeClear() PPG-65
 - IsBitRangeSet() PPG-65
 - IsBitSet() PPG-65
 - IsEmptyList() PPG-43
 - IsMemOwned() PPG-56
 - IsMemReadable() PPG-56
 - IsMemWritable() PPG-56, PPG-94
 - IsNode() PPG-44
 - IsNodeB() PPG-44
 - item ID number PPG-11
 - item number PPG-72
 - of a device PPG-106
 - item types PPG-11, PPG-69
 - ItemNode data structure PPG-68
 - items
 - absolute address of PPG-73
 - changing ownership PPG-74
 - changing priority PPG-74
 - checking existence of PPG-74
 - closing PPG-75
 - convenience calls PPG-72
 - creating PPG-67, PPG-69
 - deleting PPG-12, PPG-75
 - finding PPG-73
 - handling PPG-10
 - managing PPG-11
 - opening PPG-69, PPG-75
 - procedure for working with PPG-67
 - shared resources PPG-10
 - working with PPG-68
 - items, managing PPG-67
-

J

joystick data PPG-198
JString PPG-218

K

kernel
 high level functions PPG-1
 page allocation PPG-6
 understanding PPG-1
keyboard PPG-102
KillEventUtility() PPG-216

L

LastNode() PPG-44
LAUNCHME.M2 PPG-20
library modules
 DLL modules PPG-288
light gun
 subcode PPG-214
LightGunData PPG-197
link data structure PPG-49
linked list
 adding a node to PPG-40
 data structures PPG-46
 traversing PPG-43
linking DLLs PPG-295
list data structure PPG-48
List nodes PPG-42
ListAnchor PPG-48
ListenerList data structure PPG-210
listeners PPG-176
 disconnection PPG-206
 focus-dependent PPG-178
 focus-independent PPG-178
 focus-interested PPG-178
 list of PPG-207
 reconfiguring PPG-206
LoadGameData() PPG-276, PPG-278, PPG-283
LoadIcon() PPG-272, PPG-274
loading
 dynamic PPG-288, PPG-289
Locale structure PPG-218
 fields PPG-219
 using PPG-223
localization PPG-217
locking a resource PPG-79

LockSemaphore() PPG-12, PPG-78, PPG-79
LogEvent() PPG-320
LookupItem() PPG-11, PPG-68, PPG-73
LookupSymbol() PPG-299
Lumberjack
 creating PPG-320
 deleting PPG-321
 getting and releasing buffers PPG-321
 getting information PPG-321
 logging custom events PPG-320
 parsing buffers PPG-322
 starting and stopping PPG-320

M

main() function
 and DLLs PPG-288
malloc() PPG-5
managing items PPG-67
managing linked lists PPG-37
managing memory PPG-51, PPG-319
MEMC_GIVE PPG-57, PPG-58
MEMC_NOWRITE PPG-57
MEMC_OKWRITE PPG-57
MEMDEBUG
 compile option PPG-59
MemDebug PPG-59
MEMDEBUGF_PAD_COOKIES option PPG-61
MemInfo structure PPG-55
memory
 allocating PPG-5, PPG-53
 flags PPG-53
 getting from system-wide free memory pool PPG-58
 organization of PPG-52
 sharing PPG-9
 system size PPG-5
memory allocation PPG-5
 cause failure PPG-61
 checking for corruption PPG-60
memory allocations
 checking PPG-60
 controlling PPG-230
Memory Allocation Flags PPG-53
memory block
 continuous allocation of PPG-8
 indexing PPG-54
memory fence PPG-8

- memory information PPG-55
 - how big a block is PPG-55
 - how much avail PPG-55
 - memory management PPG-2, PPG-51–PPG-62
 - memory organization PPG-52
 - memory page
 - getting size of PPG-56
 - memory pages PPG-52
 - owning and sharing PPG-9
 - memory pointers
 - validating PPG-56
 - MEMTYPE_FILL PPG-53, PPG-55
 - MEMTYPE_NORMAL PPG-53, PPG-55, PPG-56
 - MEMTYPE_TRACKSIZE PPG-53, PPG-54, PPG-55
 - message
 - creating PPG-73
 - message port
 - creating PPG-73
 - message ports PPG-14
 - creating PPG-15, PPG-88
 - message queues PPG-15, PPG-88, PPG-92
 - Message structure PPG-89, PPG-93
 - messages PPG-14, PPG-83, PPG-87, PPG-178
 - buffered PPG-89, PPG-90, PPG-91, PPG-94, PPG-95
 - checking for PPG-92
 - configuration PPG-183, PPG-189
 - creating PPG-89
 - data block of PPG-91
 - example code PPG-96
 - forwarding PPG-96
 - interpreting PPG-15
 - passing PPG-87
 - priority PPG-88
 - pulling back PPG-94
 - receiving PPG-15, PPG-92
 - replying to PPG-15, PPG-94
 - sending PPG-15, PPG-91
 - short PPG-89, PPG-90, PPG-95
 - small PPG-91, PPG-93
 - standard PPG-89, PPG-91, PPG-93, PPG-95
 - waiting for PPG-92
 - with no reply ports PPG-95
 - working with PPG-93
 - message ports
 - finding PPG-96
 - metronome timing PPG-120, PPG-122
 - vblank PPG-122
 - microsecond clock PPG-117
 - minfo_SysFree PPG-56
 - minfo_SysLargest PPG-56
 - minfo_TaskFree PPG-56
 - MinimizeFileSystem() PPG-153
 - MinNode structure PPG-47, PPG-48
 - MkNodeID() PPG-69, PPG-73
 - MkNodeID() macro PPG-69
 - ModifyStorageReq() PPG-286
 - module symbols PPG-289
 - modules
 - DLL PPG-288
 - functions for handling PPG-296
 - library PPG-288
 - MountFileSystem() PPG-153
 - mouse PPG-102
 - monitoring PPG-215
 - return data PPG-196
 - MouseEventData data structure PPG-196
 - multi-module applications PPG-289
 - multitasking PPG-2
- ## N
-
- n_Name field PPG-47
 - n_Priority field PPG-39
 - NamelessNode structure PPG-47
 - NextNode() PPG-44
 - NextTagArg() PPG-35
 - node PPG-39
 - adding a node to the tail of a list PPG-40, PPG-41
 - adding according to alphabetical order PPG-41
 - adding according to other node values PPG-42
 - adding according to priority PPG-41
 - adding to a list PPG-40
 - adding to the head of a list PPG-40
 - changing priority of PPG-42
 - characteristics of PPG-39
 - finding by name PPG-45, PPG-46
 - priority of PPG-39
 - removing from a list PPG-42
 - node priority PPG-41
 - node structure PPG-46
 - node values PPG-42
 - non-UI events PPG-185
 - NoteTracker PPG-39
 - Notification by message PPG-106
 - Notification by signal PPG-106

NTSC PPG-120

NumericSpec structure PPG-220

O

ObtainLumberjackBuffer() PPG-321

OpenCompressionFolio() PPG-228

OpenDeviceStack() PPG-105

OpenDirectoryItem() PPG-152

OpenDirectoryPath() PPG-152

OpenFile() PPG-138, PPG-140, PPG-141

OpenFileInDir() PPG-140, PPG-142

opening a device PPG-105

OpenItem() PPG-11, PPG-75

OpenModule() PPG-20, PPG-296

OPENMODULE_FOR_TASK PPG-20

OPENMODULE_FOR_THREA PPG-20

OpenNamedDevice() PPG-116

OpenRawFile() PPG-146

P

PAL PPG-120

parent tasks PPG-4, PPG-19

ParseIFF() PPG-252, PPG-256, PPG-262

pathname PPG-140

- appending PPG-169

- determining PPG-169

- finding final component PPG-169

- specifications for PPG-140

pathnames PPG-140

PerformCopy() PPG-165, PPG-168

photo-optic gun PPG-102

pod table PPG-199

- gaining access to PPG-200

- relinquishing PPG-200

- synchronizing PPG-206

PodData data structure PPG-211

PodDescription data structure PPG-208

PodDescriptionList data structure PPG-207

Pods PPG-176

- commanding PPG-211

- listing connected PPG-207

PopChunk() PPG-254, PPG-255, PPG-269

Portfolio devices PPG-115

portfolio devices

- communicating with PPG-116

Portfolio list PPG-38

portfolio lists

- characteristics of PPG-38

PREPLIST macro PPG-40

PrepList() PPG-39

PrevNode() PPG-45

printf() PPG-311, PPG-313

PushChunk() PPG-254, PPG-255, PPG-268,
PPG-269

Q

quantum remaindering PPG-24

quantums PPG-24

QueryStorageReq() PPG-283, PPG-285

Quick I/O PPG-110

R

RationMemDebug() PPG-60, PPG-61

RawFile

- clearing errors PPG-149

- closing PPG-151

- getting info PPG-149

- opening PPG-146

- reading PPG-147

- seeking in PPG-148

- setting attributes PPG-150

- setting size PPG-148

- writing to PPG-147

RawFile functions PPG-138, PPG-145

ReadBattClock() PPG-172

ReadChunk() PPG-252, PPG-269

ReadDirectory() PPG-153

reading data

- from a file PPG-128

ReadRawFile() PPG-147

ready queue PPG-3

ready to run

- task state PPG-3

Reallocating a Block of Memory PPG-54

ReallocMem() PPG-54, PPG-55

RegisterCollectionChunks() PPG-251, PPG-258,
PPG-263

RegisterPropChunks() PPG-263

RegisterPropChunks() PPG-259, PPG-261

RegisterStopChunks() PPG-252, PPG-262,
PPG-263

REIMPORT_ALLOWED flag PPG-292

relative pathnames PPG-140

ReleaseLumberjackBuffer() PPG-321
 releasing resources PPG-4
 RemHead() PPG-42
 RemNode() PPG-43
 RemoveContextInfo() PPG-267
 removing a node PPG-42
 removing a specific node PPG-43
 removing the last node PPG-43
 RemTail() PPG-43
 Rename() PPG-141
 reply ports PPG-15, PPG-94
 ReplyMsg() PPG-94, PPG-95, PPG-96, PPG-181, PPG-191
 ReplySmallMsg() PPG-15, PPG-95, PPG-96
 Requestor folio PPG-281
 how to use PPG-282
 purpose PPG-281
 requestor object
 creating PPG-283
 resource
 locking PPG-79
 unlocking PPG-80
 resource sharing PPG-2
 returning memory PPG-9
 round-robin scheduling PPG-3, PPG-24
 running
 task state PPG-3

S

SampleSystemTimeTV() PPG-118
 SampleSystemTimeVBL() PPG-121
 SanityCheckMemDebug() PPG-60
 SaveGame
 FORM SGME chunk PPG-276
 SGDATA structure PPG-278
 SaveGame folio PPG-275, PPG-283
 functions PPG-276
 purpose PPG-275
 SaveGameData() PPG-276
 SaveIcon() PPG-272, PPG-274
 ScanList() PPG-43
 ScanListB PPG-44
 ScavengeMem() PPG-53, PPG-58
 Script folio PPG-315
 SCRIPT_TAG_BACKGROUND_MODE PPG-316
 SCRIPT_TAG_BACKGROUND_MODE tag PPG-317
 SCRIPT_TAG_CONTEXT tag PPG-316

ScriptContext structur PPG-316
 ScriptContext structure PPG-316, PPG-317
 SeekChunk() PPG-270
 SeekRawFile() PPG-148
 semaphore PPG-12
 creating PPG-73
 deleting PPG-80
 finding PPG-80
 using PPG-78
 semaphores PPG-65, PPG-77
 SendIO PPG-110
 SendIO() PPG-109, PPG-116
 SendMsg() PPG-15, PPG-91
 SendSignal() PPG-14, PPG-86
 SendSmallMsg() PPG-91
 SER_CMD_BREAK PPG-135
 SER_CMD_GETCONFIG PPG-134
 SER_CMD_SETCONFIG PPG-133
 SER_CMD_SETDTR PPG-135
 SER_CMD_SETRTS PPG-135
 SER_CMD_STATUS PPG-135
 SER_CMD_WAITEVENT PPG-134
 SerConfig structure PPG-132
 serial devices PPG-115
 serial lines
 controlling PPG-135
 DTR PPG-135
 RTS PPG-135
 serial port
 configuring PPG-133
 getting state of PPG-135
 reading configuration PPG-134
 reading data from PPG-132
 writing data to PPG-131
 SetBg comman PPG-316
 SetBitRange() PPG-64
 SetFg command PPG-316
 SetFileAttrs() PPG-142
 SetFocus data structure PPG-211
 SetItemOwner PPG-4
 SetItemOwner() PPG-23, PPG-74
 SetItemPri() PPG-11, PPG-23, PPG-74
 SetNodePri() PPG-39, PPG-42, PPG-48
 SetRawFileAttrs() PPG-150
 SetRawFileSize() PPG-148
 SGME chunk PPG-276
 shared resources
 SEE ALSO items PPG-10
 sharing memory PPG-9

sharing system resources PPG-77

SIGF_IODONE PPG-106

signal bits

allocation PPG-84

freeing PPG-86

sampling, changing PPG-86

signal masks PPG-13

signals PPG-13, PPG-83

freeing PPG-14

receiving PPG-14, PPG-85

sample code PPG-87

sending PPG-14, PPG-86

used with messages PPG-88

using PPG-84

waiting for PPG-85

sleep PPG-14

spawning tasks PPG-5

special tag commands PPG-32

stack

device PPG-105

stack size PPG-25

StartMetronome() PPG-120

StartMetronomeVBL() PPG-122

Stereoscopic glasses PPG-102

StickEventData PPG-198

Storage Manager Interface

displaying PPG-285

Storage Manager interface PPG-281

storage requestor object

deleting PPG-286

displaying PPG-285

modifying PPG-286

StoreContextInfo() PPG-253, PPG-264, PPG-266

STORREQ_CANCEL PPG-285

STORREQ_OK PPG-285

SubTimerTicks() PPG-123

SubTimes() PPG-118

Subtimes() PPG-122

symbols

DLL PPG-289

in modules PPG-289

system events PPG-185

system signals PPG-13

system time

reading the current PPG-118

system-wide free memory pool PPG-58

T

ta_Tag PPG-32

tag argument types PPG-70

tag arguments PPG-70

varArgs PPG-71

tag commands

TAG_JUMP PPG-32

TAG_NOP PPG-32

TAG_END PPG-32

TAG_ITEM_PRI PPG-21

TAG_JUMP PPG-32

TAG_NOP PPG-32

TagArg data structure PPG-70, PPG-71

TagArgs

defined PPG-32

using PPG-31

Tags

special commands PPG-32

tags PPG-31

task

defined PPG-19

task control block PPG-2, PPG-3

task priorities PPG-3

execution of PPG-3

task states PPG-3

ready to run PPG-3

running PPG-3

waiting PPG-3

tasks

changing ownership PPG-23

changing priorities PPG-23

child PPG-5

closing PPG-4

communication among PPG-26

controlling state PPG-22

creating PPG-20

ending PPG-21

intertask communications PPG-13

parent PPG-4

spawning PPG-5

starting on bootup PPG-20

thread PPG-5

yielding PPG-22

TCB PPG-2, PPG-3, PPG-14, PPG-85

TCB data structure PPG-20

thread

creating PPG-73

Threads PPG-20
 threads PPG-5, PPG-24
 ending PPG-26
 returning memory on death PPG-26
 starting PPG-24
 threadSig1 PPG-84
 threadSig2 PPG-84
 time
 reading PPG-118
 waiting for PPG-118, PPG-119
 TimeLaterThan() PPG-123
 TimeLaterThanOrEqual() PPG-123
 timer device PPG-115
 waiting for specific time PPG-118
 timer devices PPG-117
 timer IOReq
 creating PPG-117
 TIMER_CMD_GETTIME_VBL PPG-121
 TIMER_UNIT_VBLANK PPG-117
 TIMERCMD_DELAY PPG-119
 TIMERCMD_DELAY_VBL PPG-121
 TIMERCMD_DELAYUNTIL_USEC PPG-118
 TIMERCMD_DELAYUNTIL_VBL PPG-121
 TIMERCMD_METRONOME PPG-120, PPG-122
 TimerTicksLaterThan() PPG-123
 TimerTicksLaterThanOrEqual() PPG-123
 TimeVal structure PPG-117
 timing
 metronome PPG-120
 timing hardware PPG-117
 trackball PPG-102
 trackball return data PPG-196
 TRACKED_SIZE PPG-54
 transferring memory PPG-58
 traversing a linked list PPG-43
 traversing a list
 from back to front PPG-44
 from front to back PPG-43
 trigger masks PPG-176, PPG-184, PPG-188, PPG-190
 TXTR FORM PPG-245, PPG-260

U

UI events PPG-185
 UniCode PPG-218
 UnimportByAddress() PPG-298
 UnimportByName() PPG-298
 UniversalInsertNode() PPG-39, PPG-42

UNIX PPG-140
 UnloadIcon() PPG-272, PPG-274
 UnlockSemaphore() PPG-12, PPG-78, PPG-80

V

ValidateDate() PPG-174
 varargs PPG-34
 VarArgs Tag Arguments PPG-71
 vblank PPG-120
 vblank count
 reading PPG-121
 waiting for PPG-121
 vblank interval
 waiting for PPG-121
 VBlankTimeVal structure PPG-120
 vertical blank timer PPG-120

W

wait queue PPG-4
 wait state PPG-22, PPG-85
 waiting
 task state PPG-3
 waiting for signals PPG-85
 waiting queue PPG-3
 waiting tasks PPG-4
 WaitIO() PPG-4, PPG-110
 WaitPort() PPG-4, PPG-15, PPG-92, PPG-111
 WaitSignal() PPG-4, PPG-14, PPG-22, PPG-23, PPG-26, PPG-85, PPG-86, PPG-111
 WaitTime() PPG-119
 WaitTimeVBL() PPG-121
 WaitUntil () PPG-118
 WaitUntil() PPG-118
 WaitUntilVBL() PPG-121
 write permissions
 memory fence restrictions of PPG-8
 WriteBackDCache() PPG-64
 WriteBattClock() PPG-172
 WriteChunk() PPG-254, PPG-255, PPG-269
 WriteRawFile() PPG-147

Y

Yield() PPG-4, PPG-22, PPG-24
 yielding CPU time PPG-22



3DO M2 Portfolio Programmer's Reference

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

1

ANSI C Link Library Calls

printf.....	Standard C formatting routines.....	PPR-3—7
fprintf.....	Standard C formatting routines.....	PPR-3—7
sprintf.....	Standard C formatting routines.....	PPR-3—7
vprintf.....	Standard C formatting routines.....	PPR-3—7
vfprintf.....	Standard C formatting routines.....	PPR-3—7
vsprintf.....	Standard C formatting routines.....	PPR-3—7
cprintf.....	Standard C formatting routines.....	PPR-3—7
vcprintf.....	Standard C formatting routines.....	PPR-3—7

Math

abs.....	Absolute Value.....	PPR-8
labs.....	Absolute Value.....	PPR-8

2

Batt Folio Calls

ReadBattClock.....	Reads the current setting of the battery-backed clock.....	PPR-11
WriteBattClock.....	Sets the battery-backed clock.....	PPR-12

3

Compression Folio Calls

CreateCompressor.....	Creates a compression engine.....	PPR-15
CreateDecompressor.....	Creates a decompression engine.....	PPR-17
DeleteCompressor.....	Deletes a compression engine.....	PPR-19
DeleteDecompressor.....	Deletes a decompression engine.....	PPR-20
FeedCompressor.....	Gives data to a compression engine.....	PPR-21

FeedDecompressor	Gives data to a decompression engine.	PPR-22
GetCompressorWorkBufferSize	Gets the size of the work buffer needed by a compression engine.....	PPR-23
GetDecompressorWorkBufferSize	Gets the size of the work buffer needed by a decompression engine.....	PPR-24
SimpleCompress	Compresses some data in memory.....	PPR-25
SimpleDecompress.....	Decompresses some data in memory.....	PPR-26

4

Date Folio Calls

ConvertGregorianToTimeVal ...	Converts from a real-world date representation to a system TimeVal.....	PPR-29
ConvertTimeValToGregorian ...	Converts from a system TimeVal to a real-world date representation.....	PPR-30
ValidateDate.....	Makes sure a date is valid.	PPR-31

5

Debug Console Link Library Calls

CreateDebugConsole.....	Creates a view for debugging console output.	PPR-35
DebugConsoleClear.....	Erases everything in a debugging view console.	PPR-36
DebugConsoleMove.....	Moves the debugging console rendering cursor.....	PPR-37
DebugConsolePrintf.....	Output information to a debugging view console.....	PPR-38
DeleteDebugConsole.....	Closes the debugging console view preventing further debugging output..	PPR-39

6

Device Command Reference

CD-ROM

CDROMCMD_CLOSE_DRAWER	Close the CD-ROM drive's drawer.	PPR-43
CDROMCMD_DISCDATA	Requests the Disc Information (DiscID, TOC, and Session Info).	PPR-44
CDROMCMD_OPEN_DRAWER	Open the CD-ROM drive's drawer.....	PPR-45
CDROMCMD_READ	Requests specific block (sector) data from the disc.	PPR-46
CDROMCMD_READ_SUBQ	Requests the QCode data for the current sector.	PPR-50
CDROMCMD_SCAN_READ.....	Requests approximate block (sector) data from the disc.....	PPR-51
CDROMCMD_SETDEFAULTS..	Set/Get the current device defaults for read-mode IOReqs.	PPR-55

File

FILECMD_ALLOCBLOCKS	Allocate storage space for a file.	PPR-59
FILECMD_FSSTAT	Get filesystem status information.	PPR-60
FILECMD_GETPATH	Get pathname of a file.	PPR-61
FILECMD_READDIR.....	Read a directory entry by number.....	PPR-62

FILECMD_READENTRY	Read a directory entry by name.	PPR-63
FILECMD_SETEOF	Set the End Of File (EOF).....	PPR-64
FILECMD_SETTYPE	Set file type.	PPR-65
FILECMD_SETVERSION.....	Set revision and version of a file.	PPR-66

Generic

CMD_BLOCKREAD	Reads data from a block-oriented device.	PPR-67
CMD_BLOCKWRITE.....	Writes data to a block-oriented device.	PPR-68
CMD_GETMAPINFO	Returns information about the memory-mapping capabilities of a device. .	PPR-69
CMD_MAPRANGE	Requests a device to map itself into memory.	PPR-70
CMD_PREFER_FSTYPE	Requests the preferred filesystem type for a device.....	PPR-71
CMD_STATUS	Requests the DeviceStatus information for a device.	PPR-72
CMD_STREAMREAD	Reads data from a stream-oriented device.....	PPR-73
CMD_STREAMWRITE.....	Writes data to a stream-oriented device.....	PPR-74
CMD_UNMAPRANGE.....	Requests a device to unmap itself from memory.....	PPR-75

Graphics

GFXCMD_EXECUTETECMDS ..	Submits a list of rendering commands for execution by the Triangle Engine.	PPR-76
-------------------------	---	--------

Host

HOST_CMD_RECV.....	Waits for a packet of information from a remote host development system.....	PPR-78
HOST_CMD_SEND	Sends a packet of information to a remote host development system.....	PPR-79

HostConsole

HOSTCONSOLE_CMD_GETCMDLINE	Gets command-line input from the remote host.	PPR-80
----------------------------	--	--------

HostFS

HOSTFS_CMD_ALLOCBLOCKS	Controls the number of blocks allocated to a file on a remote host file system.	PPR-81
HOSTFS_CMD_BLOCKREAD ..	Reads data from an opened file on a remote host file system.....	PPR-82
HOSTFS_CMD_BLOCKWRITE	Writes data to a remote host file system.....	PPR-83
HOSTFS_CMD_CLOSEENTRY	Concludes use of a reference token.	PPR-84
HOSTFS_CMD_CREATEDIR	Creates a new directory within an existing directory.....	PPR-85

HOSTFS_CMD_CREATEFILE	Creates a new file within an existing directory.	PPR-86
HOSTFS_CMD_DELETEENTRY	Deletes an entry within a remote file system.....	PPR-87
HOSTFS_CMD_DISMOUNTFS	Requests that a file system be dismounted.....	PPR-88
HOSTFS_CMD_FSSTAT	Obtains information about a remote host file system.....	PPR-89
HOSTFS_CMD_MOUNTFS	Requests that a file system be mounted on a remote host.	PPR-90
HOSTFS_CMD_OPENENTRY ..	Obtains a reference token for an object within a remote file system.	PPR-91
HOSTFS_CMD_READDIR	Obtains information about an object within a remote host file system.....	PPR-92
HOSTFS_CMD_READENTRY	Obtains information about an object within a remote host file system.....	PPR-93
HOSTFS_CMD_RENAMEENTRY	Renames an object on a remote host file system.....	PPR-94
HOSTFS_CMD_SETBLOCKSIZE	Sets the device block size for a particular IOReq dealing with a file on a remote host file system.....	PPR-95
HOSTFS_CMD_SETEOF	Sets the logical byte count of a file on a remote host file system.....	PPR-96
HOSTFS_CMD_SETTYPE	Sets the four byte file type for an object on a remote host file system.....	PPR-97
HOSTFS_CMD_SETVERSION	Set the version and revision codes for an opened entry on a remote host file system.....	PPR-98
HOSTFS_CMD_STATUS	Obtains information about an opened entry on a remote host file system. .	PPR-99

MPEG

MPEGVIDEOCMD_CONTROL..	Permits application control over MPEG video decoding.	PPR-100
MPEGVIDEOCMD_READ	Submits a bitmap item for the MPEG video device to place a decoded picture.	PPR-102
MPEGVIDEOCMD_WRITE.....	Submits a buffer of MPEG video data for decoding by the MPEG video device.	PPR-103

Serial

SER_CMD_BREAK.....	Sends a break signal over the serial line.	PPR-104
SER_CMD_GETCONFIG	Determines the current serial port settings.	PPR-105
SER_CMD_SETCONFIG	Sets the configuration of the serial port.	PPR-106
SER_CMD_SETDTR.....	Controls the state of the serial DTR line.	PPR-107
SER_CMD_SETLOOPBACK	Controls the state of automatic serial loopback mode.	PPR-108

SER_CMD_SETRTS	Controls the state of the serial RTS line.	PPR-109
SER_CMD_STATUS	Gets the state of various serial attributes.	PPR-110
SER_CMD_WAITEVENT	Waits for serial events to occur.	PPR-111

Timer

TIMERCMD_DELAYUNTIL_USEC	Waits for a given time.	PPR-112
TIMERCMD_DELAYUNTIL_VBL	Waits for the system VBL count to reach a specific number.	PPR-113
TIMERCMD_DELAY_USEC	Waits for a fixed amount of time to pass.	PPR-114
TIMERCMD_DELAY_VBL	Waits for a fixed number of vertical blanking intervals.	PPR-115
TIMERCMD_GETTIME_USEC	Returns the current system time.	PPR-116
TIMERCMD_GETTIME_VBL	Returns the current system time in vertical blanking intervals.	PPR-117
TIMERCMD_METRONOME_USEC	Requests to be signalled at regular intervals of time.	PPR-118
TIMERCMD_METRONOME_VBL	Requests to be signalled at regular intervals of time.	PPR-119
TIMERCMD_SETTIME_USEC	Sets the current system time.	PPR-120
TIMERCMD_SETTIME_VBL	Sets the current system time in vertical blanking intervals.	PPR-121

7

Event Broker Calls

GetControlPad	Gets control pad events.	PPR-125
GetMouse	Gets mouse events.	PPR-126
InitEventUtility	Connects task to the event broker.	PPR-127
KillEventUtility	Disconnects a task from the event broker.	PPR-128

8

File Folio Calls

Directory

ChangeDirectory	Changes the current directory.	PPR-131
ChangeDirectoryInDir	Changes the current directory relative to another directory.	PPR-132
CloseDirectory	Closes a directory.	PPR-133
CreateDirectory	Creates a directory.	PPR-134
CreateDirectoryInDir	Creates a directory relative to another directory.	PPR-135
DeleteDirectory	Deletes a directory.	PPR-136

DeleteDirectoryInDir	Deletes a directory relative to another directory.....	PPR-137
GetDirectory.....	Gets the item number and pathname for the current directory.	PPR-138
OpenDirectoryItem.....	Opens a directory specified by an item.	PPR-139
OpenDirectoryPath.....	Opens a directory specified by a pathname.....	PPR-140
ReadDirectory	Reads the next entry from a directory.....	PPR-141

File

CloseFile	Closes a file.....	PPR-143
CreateAlias	Creates a file system alias.....	PPR-144
CreateFile	Creates a file.	PPR-145
CreateFileInDir	Creates a file relative to a directory.	PPR-146
DeleteFile	Deletes a file.	PPR-147
DeleteFileInDir.....	Deletes a file relative to a directory.	PPR-148
FindFileAndIdentify	Searches one or more locations for a file, and returns its pathname.	PPR-149
FindFileAndOpen	Searches one or more locations for a file, and opens the file.....	PPR-150
OpenFile.....	Opens a disk file.....	PPR-151
OpenFileInDir	Opens a disk file relative to a directory.	PPR-152
Rename	Rename a file, directory, or filesystem.....	PPR-153
SetFileAttrs	Sets some attributes of a file.	PPR-154

RawFile

ClearRawFileError	Clears the error state of a file.	PPR-155
CloseRawFile	Concludes access to a file.	PPR-156
GetRawFileInfo.....	Gets some information about an opened file.....	PPR-157
OpenRawFile	Gains access to a file for raw file I/O.....	PPR-158
OpenRawFileInDir().....	Gains access to a file for raw file I/O.....	PPR-160
ReadRawFile	Reads data from a file.	PPR-162
SeekRawFile.....	Moves the file cursor within a file.	PPR-164
SetRawFileAttrs	Sets some attributes of an opened file.	PPR-166
SetRawFileSize.....	Sets the size of an opened file.....	PPR-168
WriteRawFile.....	Writes data to a file.	PPR-169

9

FileSystem Utilities Folio Calls

AppendPath	Appends a path to an existing path specification.	PPR-173
CreateCopyObj	Creates a copier object to perform hierarchical copy operations.	PPR-174
DeleteCopyObj	Releases any resources consumed by \f4CreateCopyObj()\fP.....	PPR-177
DeleteTree.....	Deletes a complete filesystem directory tree.....	PPR-178
FindFinalComponent	Returns the last component of a path.....	PPR-180

GetPath.....	Obtain the absolute directory path leading to an opened file.....	PPR-181
PerformCopy	Enter the copier engine and start copying files.	PPR-182

10

Icon Folio Calls

LoadIcon.....	Loads icon data into memory and prepares it for rendering.	PPR-185
SaveIcon	Creates an IFF Icon file.....	PPR-187
UnloadIcon	Unloads an icon from memory and frees any resources associated with it.	PPR-189

11

IFF Folio Calls

AllocContextInfo	Allocates and initializes a ContextInfo structure.....	PPR-193
AttachContextInfo	Attaches a ContextInfo structure to a given ContextNode.	PPR-194
CreateIFFParser	Creates a new parser structure and prepares it for use.....	PPR-195
DeleteIFFParser	Deletes an IFF parser.	PPR-197
FindCollection	Gets a pointer to the current list of collection chunks.	PPR-198
FindContextInfo	Returns a ContextInfo structure from the context stack.....	PPR-199
FindPropChunk	Searches for a stored property chunk.	PPR-200
FindPropContext	Gets the parser's current property context.	PPR-201
FreeContextInfo	Deallocate a ContextInfo structure.....	PPR-202
GetCurrentContext	Gets a pointer to the ContextNode for the current chunk.	PPR-203
GetIFFOffset.....	Returns the absolute seek position within the current IFF stream.....	PPR-204
GetParentContext.....	Gets the nesting ContextNode for the given ContextNode.....	PPR-205
InstallEntryHandler	Adds an entry handler to the parser.	PPR-207
InstallExitHandler.....	Adds an exit handler to the parser.	PPR-208
ParseIFF.....	Parses an IFF stream.	PPR-210
PopChunk	Pops the top context node off the context stack.	PPR-212
PushChunk	Pushes a new context node on the context stack.....	PPR-213
ReadChunk	Reads bytes from the current chunk into a buffer.....	PPR-214
ReadChunkCompressed.....	Reads and decompresses bytes from the current chunk into a buffer.	PPR-215
RegisterCollectionChunks.....	Defines chunks to be collected during the parse operation.....	PPR-216
RegisterPropChunks.....	Defines property chunks to be stored during the parse operation.	PPR-217
RegisterStopChunks	Defines chunks that cause the parser to stop and return to the caller.....	PPR-218
RemoveContextInfo	Removes a ContextInfo structure from wherever it is attached.....	PPR-219
SeekChunk.....	Moves the current position cursor within the current chunk.....	PPR-220
StoreContextInfo.....	Inserts a ContextInfo structure into the context stack.....	PPR-222

WriteChunk	Writes data from a buffer into the current chunk	PPR-223
WriteChunkCompressed	Compressed data from a buffer and writes the result to the current chunk.....	PPR-224

12

International Folio Calls

intlCloseLocale.....	Terminates use of a given Locale item.....	PPR-227
intlCompareStrings	Compares two strings for collation purposes.	PPR-228
intlConvertString.....	Changes certain attributes of a string.	PPR-229
intlFormatDate	Formats a date in a localized manner.....	PPR-231
intlFormatNumber.....	Format a number in a localized manner.	PPR-233
intlGetCharAttrs	Returns attributes describing a given character.....	PPR-236
intlLookupLocale	Returns a pointer to a Locale structure.....	PPR-238
intlOpenLocale	Gains access to a Locale item.....	PPR-239
intlTransliterateString	Converts a string between character sets.	PPR-240

13

JString Folio Calls

ConvertASCII2ShiftJIS	Converts an ASCII string to Shift-JIS.....	PPR-245
ConvertFullKana2HalfKana.....	Convert a full-width kana string to a half-width kana string.	PPR-246
ConvertFullKana2Hiragana.....	Convert a full-width kana string to a hiragana string.....	PPR-247
ConvertFullKana2Romaji.....	Convert a full-width kana string to a romaji string.	PPR-248
ConvertHalfKana2FullKana.....	Convert a half-width kana string to a full-width kana string.	PPR-249
ConvertHalfKana2Hiragana	Convert a half-width kana string to a hiragana string.....	PPR-250
ConvertHalfKana2Romaji	Convert a half-width kana string to a romaji string.	PPR-251
ConvertHiragana2FullKana.....	Convert a hiragana string to a full-width kana string.....	PPR-252
ConvertHiragana2HalfKana	Convert a hiragana string to a half-width kana string.....	PPR-253
ConvertHiragana2Romaji	Convert a hiragana string to a romaji string.....	PPR-254
ConvertRomaji2FullKana.....	Convert a romaji string to a full-width kana string.	PPR-255
ConvertRomaji2HalfKana	Convert a romaji string to a half-width kana string.	PPR-256
ConvertRomaji2Hiragana	Convert a romaji string to a hiragana string.....	PPR-257
ConvertShiftJIS2ASCII	Converts a Shift-JIS string to ASCII.....	PPR-258
ConvertShiftJIS2Unicode	Converts a Shift-JIS string to UniCode.	PPR-259
ConvertUnicode2ShiftJIS	Converts a UniCode string to Shift-JIS.	PPR-260

14 Kernel Folio Calls

BitArrays

AtomicClearBits	Clears bit in a word of memory in an atomic manner.	PPR-263
AtomicSetBits	Sets bit in a word of memory in an atomic manner.	PPR-264
ClearBitRange	Clear a range of bits within a bit array.	PPR-265
CountBits	Count the number of bits set in a word.	PPR-266
DumpBitRange	Display a range of bits to the debugging terminal.	PPR-267
FindClearBitRange	Find a range of clear bits within a bit array.	PPR-268
FindLSB	Finds the least-significant bit.	PPR-269
FindMSB	Finds the highest-numbered bit.	PPR-270
FindSetBitRange	Find a range of set bits within a bit array.	PPR-271
IsBitClear	Test whether a bit within a bit array is set to 0.	PPR-272
IsBitRangeClear	Test whether all bits within a range of a bit array are all set to 0.	PPR-273
IsBitRangeSet	Test whether all bits within a range of a bit array are all set to 1.	PPR-274
IsBitSet	Test whether a bit within a bit array is set to 1.	PPR-275
SetBitRange	Set a range of bits within a bit array.	PPR-276

Caches

FlushDCache	Write back modified contents of the data cache to memory, and remove the data from the cache.	PPR-277
FlushDCacheAll	Write back modified contents of the data cache to memory, and remove the data from the cache.	PPR-278
GetCacheInfo	Gets information about the CPU caches.	PPR-279
GetDCacheFlushCount	Obtain the current system cache flush count.	PPR-280
WriteBackDCache	Write back modified contents of the data cache to memory.	PPR-281

DDF

ScanForDDFToken	Scan for a particular token in a token sequence.	PPR-282
-----------------------	---	---------

Debugging

ControlIODebug	Controls what IODebug does and doesn't do.	PPR-283
DebugBreakpoint	Trigger a breakpoint-like event in the debugger.	PPR-284
DebugPutChar	Output a character to the debugging terminal.	PPR-285
DebugPutStr	Output a string to the debugging terminal.	PPR-286
GetSysErr	Gets the error string for an error.	PPR-287
PrintfSysErr	Prints the error string for an error.	PPR-288

Devices

CloseDeviceStack.....	Closes a stack of devices.	PPR-289
CreateDeviceStackList.....	Get a list of device stacks which support the requested capabilities.	PPR-290
DeleteDeviceStackList.....	Destroy the List created by \f4CreateDeviceStackList()\fP.	PPR-292
DeviceStackIsIdentical	Compare a DeviceStack to an open device stack.	PPR-293
OpenDeviceStack	Opens a stack of devices.....	PPR-294

I/O

AbortIO	Aborts an I/O operation.....	PPR-295
CheckIO	Checks whether an I/O operation is complete.	PPR-296
CreateIOReq.....	Creates an I/O request.	PPR-297
DeleteIOReq.....	Deletes an I/O request.....	PPR-299
DoIO	Performs synchronous I/O.....	PPR-300
SendIO	Requests I/O be performed.....	PPR-301
WaitIO.....	Waits for an I/O operation to complete.	PPR-303

Items

CheckItem.....	Checks to see if an item exists.....	PPR-304
CloseItem.....	Closes a previously opened item.....	PPR-305
CreateItem	Creates an item.....	PPR-306
DeleteItem	Deletes an item.....	PPR-307
FindAndOpenItem	Finds an item by type and tags and opens it.....	PPR-308
FindAndOpenNamedItem	Finds an item by name and opens it.....	PPR-309
FindItem.....	Finds an item by type and tags.....	PPR-310
FindNamedItem	Finds an item by name.....	PPR-311
IsItemOpened	Determines whether a task or thread has opened a given item.....	PPR-312
LookupItem	Gets a pointer to an item.....	PPR-313
MkNodeID.....	Assembles an item type value.....	PPR-314
OpenItem	Opens an item.....	PPR-315
SetItemOwner	Changes the owner of an item.....	PPR-316
SetItemPri.....	Changes the priority of an item.....	PPR-317

Lists

AddHead	Adds a node to the head of a list.....	PPR-318
AddTail.....	Adds a node to the tail of a list.....	PPR-319
DumpNode.....	Prints contents of a node.....	PPR-320
FindNamedNode	Finds a node by name.....	PPR-321
FindNodeFromHead	Returns a pointer to a node appearing at a given ordinal position from the head of the list.....	PPR-322

FindNodeFromTail.....	Returns a pointer to a node appearing at a given ordinal position from the tail of the list.....	PPR-323
FirstNode	Gets the first node in a list.....	PPR-324
GetNodeCount	Counts the number of nodes in a list.....	PPR-325
GetNodePosFromHead.....	Gets the ordinal position of a node within a list, counting from the head of the list.....	PPR-326
GetNodePosFromTail	Gets the ordinal position of a node within a list, counting from the tail of the list.....	PPR-327
InsertNodeAfter	Inserts a node into a list after another node already in the list.....	PPR-328
InsertNodeAlpha	Inserts a node into a list according to alphabetical order.....	PPR-329
InsertNodeBefore.....	Inserts a node into a list before another node already in the list.....	PPR-330
InsertNodeFromHead.....	Inserts a node into a list.....	PPR-331
InsertNodeFromTail	Inserts a node into a list.....	PPR-332
IsEmptyList.....	Checks whether a list is empty.....	PPR-333
IsListEmpty.....	Checks whether a list is empty.....	PPR-334
IsNode	Validates a node when moving forward through a list.....	PPR-335
IsNodeB	Validates a node when moving backward through a list.....	PPR-336
LastNode	Gets the last node in a list.....	PPR-337
NextNode.....	Gets the next node in a list.....	PPR-338
PrepList	Initializes a list.....	PPR-339
PrevNode.....	Gets the previous node in a list.....	PPR-340
RemHead.....	Removes the first node from a list.....	PPR-341
RemNode.....	Removes a node from a list.....	PPR-342
RemTail	Removes the last node from a list.....	PPR-343
ScanList.....	Walks through all the nodes in a list.....	PPR-344
ScanListB.....	Walks through all the nodes in a list backwards.....	PPR-345
SetNodePri.....	Changes the priority of a list node.....	PPR-346
UniversalInsertNode	Inserts a node into a list.....	PPR-347

Loader

CloseModule.....	Concludes use of a module item.....	PPR-348
ExecuteModule	Executes code in a module item as a subroutine.....	PPR-349
ExpungeByAddress.....	Removes a symbol from the list of exports.....	PPR-350
ExpungeBySymbol.....	Removes a symbol from the list of exports.....	PPR-351
ImportByAddress.....	Loads an exporting module based on an imported symbol.....	PPR-352
ImportByName.....	Loads and resolves the named exporting module.....	PPR-353
LookupSymbol.....	Returns the address of a loaded symbol.....	PPR-354
OpenModule	Opens an executable file from disk and prepares it for use.....	PPR-355

UnimportByAddress.....	Unloads an exporting module, based on the address of an import.	PPR-356
UnimportByName	Unloads a named module.	PPR-357

Lumberjack

ControlLumberjack	Determine which event types Lumberjack logs.	PPR-358
CreateLumberjack	Initializes Lumberjack, the Portfolio logging service.	PPR-359
DeleteLumberjack	Disables Lumberjack, the Portfolio logging service.	PPR-360
DumpLumberjackBuffer	Parse and display the contents of a Lumberjack buffer.....	PPR-361
LogEvent	Add a custom event to the Lumberjack logs.	PPR-362
ObtainLumberjackBuffer	Obtain a pointer to a Lumberjack buffer which is currently full.	PPR-363
ReleaseLumberjackBuffer	Return a logging buffer to the Lumberjack system so it can be reused. ...	PPR-364

Memory

AllocMem	Allocates a block of memory.....	PPR-365
AllocMemAligned	Allocates a block of memory aligned to a particular boundary.....	PPR-367
AllocMemMasked	Allocates a block of memory with specific bits set or cleared in its address.	PPR-369
AllocMemPages	Allocates whole pages of memory.	PPR-371
AllocMemTrack	Allocates a block of memory with size-tracking.	PPR-373
AllocMemTrackWithOptions	Allocates a block of memory with size-tracking and other options.	PPR-374
ControlMem	Controls memory permissions and ownership.	PPR-375
ControlMemDebug	Controls what MemDebug does and doesn't do.	PPR-377
CreateMemDebug	Initializes MemDebug, the Portfolio memory debugging package.	PPR-379
DeleteMemDebug	Releases memory debugging resources.	PPR-381
DumpMemDebug	Dumps memory allocation debugging information.	PPR-382
FreeMem	Frees memory that was allocated with <code>\f4AllocMem()</code> or <code>\f4AllocMemAligned()</code>	PPR-383
FreeMemPages	Frees memory that was allocated with <code>\f4AllocMemPages()</code>	PPR-384
FreeMemTrack	Frees memory that was allocated with <code>\f4AllocMemTrack()</code> or <code>\f4AllocMemTrackWithOptions()</code>	PPR-385
GetMemInfo	Gets information about available memory.....	PPR-386
GetMemTrackSize	Gets the size of a block of memory allocated with <code>MEMTYPE_TRACKSIZE</code>	PPR-388
GetPageSize	Gets the size in bytes of a page of memory.	PPR-389
IsMemOwned	Determines whether a region of memory is owned by the current task. ...	PPR-390
IsMemReadable	Determines whether a region of memory is fully readable by the current task.	PPR-391
IsMemWritable	Determines whether a region of memory is fully writable by the current task.	PPR-392
RationMemDebug	Rations memory allocations to test failure paths.	PPR-393

ReallocMem.....	Reallocates a block of memory to a different size.....	PPR-395
SanityCheckMemDebug.....	Checks all current memory allocations to make sure all the allocation cookies are intact.....	PPR-397
ScavengeMem.....	Returns the current task's unused memory pages to the system page pool.....	PPR-398

Messaging

CreateBufferedMsg.....	Creates a buffered message.....	PPR-399
CreateMsg.....	Creates a standard message.....	PPR-400
CreateMsgPort.....	Creates a message port.....	PPR-401
CreateSmallMsg.....	Creates a small message.....	PPR-402
CreateUniqueMsgPort.....	Creates a message port with a unique name.....	PPR-403
DeleteMsg.....	Deletes a message.....	PPR-404
DeleteMsgPort.....	Deletes a message port.....	PPR-405
FindMsgPort.....	Finds a message port by name.....	PPR-406
GetMsg.....	Gets a message from a message port.....	PPR-407
GetThisMsg.....	Gets a specific message.....	PPR-408
ReplyMsg.....	Sends a reply to a message.....	PPR-410
ReplySmallMsg.....	Sends a reply to a small message.....	PPR-412
SendMsg.....	Sends a message.....	PPR-413
SendSmallMsg.....	Sends a small message.....	PPR-415
WaitPort.....	Waits for a message to arrive at a message port.....	PPR-416

Miscellaneous

ReadHardwareRandomNumber.....	Gets a 32-bit random number.....	PPR-417
ReadUniqueID.....	Gets the system unique id.....	PPR-418

Semaphores

CreateSemaphore.....	Creates a semaphore.....	PPR-419
CreateUniqueSemaphore.....	Creates a semaphore with a unique name.....	PPR-420
DeleteSemaphore.....	Deletes a semaphore.....	PPR-421
FindSemaphore.....	Finds a semaphore by name.....	PPR-422
LockSemaphore.....	Locks a semaphore.....	PPR-423
UnlockSemaphore.....	Unlocks a semaphore.....	PPR-425

Signals

AllocSignal.....	Allocates signals.....	PPR-426
ClearCurrentSignals.....	Clears some received signal bits.....	PPR-428
FreeSignal.....	Frees signals.....	PPR-429

GetCurrentSignals	Gets the currently received signal bits.	PPR-430
GetTaskSignals	Gets the currently received signal bits for a task.	PPR-431
SendSignal	Sends signals to another task.	PPR-432
WaitSignal	Waits until any of a set of signals are received.	PPR-433

Tags

ConvertFP_TagData	Store a floating point value in a TagArg.	PPR-434
ConvertTagData_FP	Extract a floating point value stored in a TagArg.	PPR-435
DumpTagList	Prints the contents of a tag list.	PPR-436
FindTagArg	Looks through a tag list for a specific tag.	PPR-437
GetTagArg	Finds a TagArg in list and returns its ta_Arg field.	PPR-438
NextTagArg	Finds the next TagArg in a tag list.	PPR-439

Tasks

CreateModuleThread	Creates a thread from a loaded code module.	PPR-441
CreateTask	Creates a task from a loaded code module.	PPR-443
CreateThread	Creates a thread.	PPR-445
DeleteModuleThread	Deletes a thread.	PPR-448
DeleteTask	Deletes a task.	PPR-449
DeleteThread	Deletes a thread.	PPR-450
exit	Exits from a task or thread.	PPR-451
FindTask	Finds a task by name.	PPR-452
InvalidateFPState	Invalidates the current contents of the floating-point registers to prevent them from being saved during a context switch.	PPR-453
RegisterUserException	Registers the exceptions to be handled by the current task.	PPR-454
RegisterUserExcHandler	Registers an exception handler for the current task.	PPR-455
Yield	Give up the CPU to a task of equal priority.	PPR-457

Timer

AddTimerTicks	Adds two TimerTicks values together.	PPR-459
AddTimes	Adds two time values together.	PPR-460
CompareTimerTicks	Compares two timer ticks values.	PPR-461
CompareTimes	Compares two time values.	PPR-462
ConvertTimerTicksToTimeVal	Convert a hardware dependant timer tick value to a TimeVal.	PPR-463
ConvertTimeValToTimerTicks	Convert a time value to a hardware dependant form.	PPR-464
CreateTimerIOReq	Creates a timer device I/O request.	PPR-465
DeleteTimerIOReq	Delete a timer device I/O request.	PPR-466
SampleSystemTimeTT	Samples the system time with very low overhead.	PPR-467

SampleSystemTimeTV	Samples the system time with very low overhead.	PPR-468
SampleSystemTimeVBL	Samples the system VBL count with very low overhead.	PPR-469
StartMetronome	Start a metronome counter.	PPR-470
StartMetronomeVBL	Start a metronome counter.	PPR-471
StopMetronome	Stop a metronome counter.	PPR-472
StopMetronomeVBL	Stop a metronome counter.	PPR-473
SubTimerTicks	Subtracts one TimerTicks value from another.	PPR-474
SubTimes	Subtracts one time value from another.	PPR-475
TimeLaterThan	Returns whether a time value comes before another.	PPR-476
TimeLaterThanOrEqual	Returns whether a time value comes before or at the same time as another.	PPR-477
TimerTicksLaterThan	Returns whether a time tick value comes before another.	PPR-478
TimerTicksLaterThanOrEqual	Returns whether a time tick value comes before or at the same time as another.	PPR-479
WaitTime	Waits for a given amount of time to pass.	PPR-480
WaitTimeVBL	Waits for a given number of video fields to pass.	PPR-481
WaitUntil	Waits for a given amount of time to arrive.	PPR-482
WaitUntilVBL	Waits for a given vblank count to be reached.	PPR-483

15

Portfolio Item Reference

Alias	A character string for referencing the pathname to a file or a directory of files.	PPR-487
Attachment	The binding of a \f4Sample\VP or an \f4Envelope\VP to an \f4Instrument\VP or \f4Template\VP.	PPR-488
AudioClock	Audio virtual timer item.	PPR-491
Bitmap	An Item describing an image buffer in RAM.	PPR-492
Cue	Audio asynchronous notification item.	PPR-495
Envelope	Audio envelope.	PPR-496
ErrorText	A table of error messages.	PPR-501
File	A handle to a data file.	PPR-502
Instrument	DSP Instrument Item.	PPR-503
IOReq	An item used to communicate between a task and an I/O device.	PPR-506
Knob	An item for adjusting an \f4Instrument\VP's parameters.	PPR-507
Locale	A database of international information.	PPR-509
Message	The means of sending data from one task to another.	PPR-510
MsgPort	An item through which a task receives messages.	PPR-512
Probe	An item to permit the CPU to read the output of a DSP \f4Instrument\VP.	PPR-513

Projector	An Item representing a particular physical display type.	PPR-514
Sample	A digital recording of a sound.	PPR-519
Semaphore	An item used to arbitrate access to shared resources.	PPR-523
Task	An executable context.	PPR-524
TEContext	An Item describing the context of the Triangle Engine.	PPR-526
Template	A description of an audio instrument.	PPR-528
Tuning	An item that contains information for tuning an \f4Instrument\fp.	PPR-529
View	An Item describing a displayed region of imagery.	PPR-531
ViewList	An anchor for a heirarchy of sub-Views.	PPR-534

16

Requester Folio Calls

CreateStorageReq	Create a storage requester object.	PPR-537
DeleteStorageReq	Delete a storage requester object.	PPR-539
DisplayStorageReq	Display a storage requester object to the user.	PPR-540
ModifyStorageReq	Modify attributes of a storage requester object.	PPR-541
QueryStorageReq	Query the current state of certain attributes of a storage requester object.	PPR-543

17

SaveGame Folio Calls

LoadGameData	Loads a saved game file into memory.	PPR-547
SaveGameData	Stores a saved game file.	PPR-549

18

Script Folio Calls

CreateScriptContext	Creates and initializes a ScriptContext structure.	PPR-553
DeleteScriptContext	Deletes a ScriptContext structure.	PPR-554
ExecuteCmdLine	Executes a shell command-line.	PPR-555

19

Single Precision Math Library Calls

Conversion

ceilf	round towards positive infinity.	PPR-559
fabsf	Floating Point Absolute Value.	PPR-560
floorf	round towards negative infinity.	PPR-561
fmodf	get the remainder of a division.	PPR-562
modff	get integer and fractional parts of a float.	PPR-563

Power

expf.....	Exponential Function.....	PPR-564
frexpf.....	get fractional and exponent parts of a float.....	PPR-565
ldexpf.....	multiply a float by a power of 2.....	PPR-566
log10f.....	Base-10 Logarithm.....	PPR-567
logf.....	Natural Logarithm.....	PPR-568
powf.....	Power Function.....	PPR-569
rsqrtf.....	Reciprocal Square Root Functions.....	PPR-570
rsqrtff.....	Reciprocal Square Root Functions.....	PPR-570
rsqrtfff.....	Reciprocal Square Root Functions.....	PPR-570
sqrtrf.....	Square Root Functions.....	PPR-571
sqrtrff.....	Square Root Functions.....	PPR-571
sqrtrfff.....	Square Root Functions.....	PPR-571

Trigonometric

acosf.....	Inverse Trigonometric Functions.....	PPR-572
asinf.....	Inverse Trigonometric Functions.....	PPR-572
atanf.....	Inverse Trigonometric Functions.....	PPR-572
atan2f.....	arctangent of y/x.....	PPR-573
cosf.....	Cosine and Sine Functions.....	PPR-574
sinf.....	Cosine and Sine Functions.....	PPR-574
coshf.....	Hyperbolic Functions.....	PPR-575
sinhf.....	Hyperbolic Functions.....	PPR-575
tanhf.....	Hyperbolic Functions.....	PPR-575

Preface

About This Book

3DO M2 Programmer's Reference contains the command and call descriptions for the 3DO M2 Portfolio operating system.

About the Audience

This book is written for programmers and application developers. To use this document, you should have a working knowledge of the C programming language, and object- oriented concepts.

How This Book Is Organized

This manual is a command reference. It contains examples of M2 Portfolio procedure calls, syntax, and use. It contains the following chapters:

Chapter 1, ANSI C Link Library Calls —This section presents the reference documentation for some of the ANSI C functions provided as part of the 3DO development environment.

Chapter 2, Batt Folio Calls —This section presents the reference documentation for the Batt folio, which provides access to the system's battery-backed clock and memory.

Chapter 3, Compression Folio Calls —This section presents the reference documentation for the Compression folio and associated link libraries.

Chapter 4, Date Folio Calls —This section presents the reference documentation for the Date folio, which provides conversion functions to handle date and time values.

Chapter 5, Debug Console Link Library Calls —This section presents the reference documentation for the debug console link library, which provides services to perform console-style output to a 3DO View.

Chapter 6, Device Command Reference—This section presents the reference documentation for the many I/O commands supported by the various devices in the Portfolio architecture.

Chapter 7, Event Broker Calls —This section presents the reference documentation for the Event Broker and associated link libraries.

Chapter 8, File Folio Calls —This section presents the reference documentation for the File folio and associated link libraries.

Chapter 9, FileSystem Utilities Folio Calls —This section presents the reference documentation for the FSUtils folio, which provides high-level functions to perform various file system operations.

Chapter 10, Icon Folio Calls —This section presents the reference documentation for the Icon folio, which provides icon management functions.

Chapter 11, IFF Folio Calls —This section presents the reference documentation for the IFF folio and associated link libraries, which provides services to read and write files in the IFF format.

Chapter 12, International Folio Calls —This section presents the reference documentation for the International folio and associated link libraries.

Chapter 13, JString Folio Calls —This section presents the reference documentation for the JString folio and associated link libraries.

Chapter 14, Kernel Folio Calls —This section presents the reference documentation for the Kernel folio and associated link libraries.

Chapter 15, Portfolio Item Reference—This section presents the reference documentation for the various types of items supported within Portfolio. Each item type is listed along with the tags and functions that can be used to manipulate it.

Chapter 16, Requester Folio Calls —This section presents the reference documentation for the Requester folio, which provides file management user-interface components.

Chapter 17, SaveGame Folio Calls —This section presents the reference documentation for the SaveGame folio, which provides simple methods for loading and saving compressed saved game files to storage.

Chapter 18, Script Folio Calls —This section presents the reference documentation for the Script folio, which provides services to execute simple scripts.

Chapter 19, Single Precision Math Library Calls —This section presents the reference documentation for the floating point math functions provided as part of the 3DO development environment.

Related Documentation

The following manuals are useful to developers who are programming in the 3DO environment:

3DO M2 Portfolio Programmer's Guide. A guide to the features of the kernel, IO, filesystem, and so on in the 3DO operating system.

3DO M2 Audio and Music Programmer's Guide. Describes the music and audio folios. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Graphics Programmer's Guide. Describes the function calls for the M2 graphics architecture, folios, and APIs.

3DO M2 Graphics Programmer's Reference. Describes the M2 graphics architecture and APIs.. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Audio and Music Programmer's Guide. Describes the music and audio folios. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Debugger Programmer's Guide. A guide to using the 3DO Debugger. It provides a tutorial on using the debugger as well as descriptions of the graphical user interface.

3DO M2 DataStreamer Programmer's Guide. A guide to the 3DO DataStreamer architecture, streaming tools, and weaver tool.

3DO M2 DataStreamer Programmer's Reference. Provides manual pages for the structures in the 3DO DataStreamer library, for all data preparation tools, and for all weaver script commands.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>int32 OpenGraphicsFolio(void)</code>
procedure name	<code>CreateScreenGroup()</code>
new term or emphasis	A <i>ViewList</i> is a special case of a View.
file or folder name	The <i>remote</i> folder, the <i>demo.scr</i> file.

Chapter 1

ANSI C Link Library Calls

This section presents the reference documentation for some of the ANSI C functions provided as part of the 3DO development environment.


```
int printf(const char *format, ...);

int fprintf(FILE *file, const char *format, ...);

int sprintf(char *s, const char *format, ...);

int cprintf(const char *format, OutputFunc of, void *userData, ...
);

int vprintf(const char *format, va_list a);

int vfprintf(FILE *file, const char *format, va_list a);

int vsprintf(char *s, const char *format, va_list a);

int vcprintf(const char *format, OutputFunc of, void *userData,
             va_list va);
```

`printf()` converts a formatting string and a series of arguments to a string of ASCII characters which is output to the debugging terminal.

The format argument points to a string containing characters to output, along with special conversion specifications. These conversion specs indicate how to interpret the extra arguments given to this function. The specs are replaced in the output string by arguments converted according to the specifications.

A conversion specification begins with a percent sign (%). To place an ordinary percent into the output stream, precede it with another percent in the format string. That is, %% will send a single percent character to the output stream. A conversion specification has the following format:

The flags field controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes. The following flags may be specified in any order:

Causes the result to be left-justified within the field specified by width or within the default width.

Causes a plus or minus sign to be placed before the result. This flag is used in conjunction with the various numeric conversion types. If it is absent, the sign character is generated only

for a negative number.

<blank>

Causes a leading blank for a positive number and a minus sign for a negative number. This flag is similar to the plus (+) flag. If both the plus (+) and the blank flags are present, the plus (+) takes precedence.

(pound)

Causes special formatting. With the %o, %x, and %X types, this flag prefixes any output with 0, 0x, or 0X, respectively. With the %p type, this flag prefixes any output with @. With the %f, %e, and %E types, this flag forces the result to contain a decimal point. With the %g and %G types, this flag forces the result to contain a decimal point and retain trailing zeroes.

0 (zero)

Pads the field width with leading zeroes instead of spaces for the %d, %i, %o, %u, %x, %X, %e, %E, %f, %g, and %G conversion types. If the minus (-) flag is also used, the zero flag is ignored. If a precision is specified, this flag is ignored for conversion types %d, %i, %o, %u, %x, and %X.

The width is a number ≥ 0 that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right, depending on the presence of the minus (-) flag. A blank is used as the padding character unless the 0 flag was specified, in which case padding with zeroes is performed. If the minus (-) flag appears, padding is performed with blanks.

If you do not want to specify the field width as a constant in the format string, you can code it as an asterisk (*). The asterisk indicates that the width value is a 32-bit integer in the argument list.

The precision value specifies the field precision, which is the required precision of floating-point conversions, the maximum number of characters to be copied from a string, or the minimum number of digits to output for integer conversions.

The meaning of the precision value depends on the conversion type. For %c, the precision value is ignored. For %d, %i, %o, %u, %x, and %X, the precision value specifies the minimum number of digits to output, and zero padding to the left is supplied if fewer digits are generated. For %e, %E, and %f the precision is the number of digits to appear after the decimal point. If fewer digits are generated, trailing zeroes are supplied. For %g and %G, the precision is the maximum number of significant digits. Finally, for %s, the precision is the maximum number of characters to be copied from the string.

As with the width value, you can use an asterisk (*) for the precision to indicate that the value should be picked up from the argument list.

The type field is a one or two letter sequence specifying the argument conversion to perform. The possible values are:

c

Specifies a single-character conversion. The associated argument is a 32-bit integer. The low byte of this integer is output.

d

Specifies a signed decimal integer conversion. The associated argument is a signed 32-bit integer, and the result is a string of digits preceded by a sign. If the plus (+) and blank flags are absent, the sign is produced only for a negative integer.

e

Specifies a floating-point conversion. The associated argument is a floating-point number, and the result has the form:

`[-]s.dddddde[+/-]ee`

where *s* is a single decimal digit, *dddddd* is one or more digits, and *ee* is an exponent of at least two digits. The plus (+) and blank flags dictate whether there will be a leading sign character emitted if the number is positive. One digit is output before the decimal point, and the number of digits output after the decimal point depends on the requested precision. The value is rounded to the specified number of digits. If no precision is specified, the default is six decimal places.

E

Is identical to `%e`, except that the result has the form:

`[-]s.dddddE[+/-]ee`

f

Specifies a floating-point conversion. The associated argument is a floating-point number, and the result has the form:

`[-]sss.dddddd`

where *sss* indicates one or more decimal digits. The minus sign is omitted if the number is positive, but a sign character will still be generated if the plus (+) or blank flag is present. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits output after the decimal point depends on the requested precision. If no precision is specified, the default is six decimal places. If the precision is specified as 0, or if there are no nonzero digits to the right of the decimal point, then the decimal point is omitted unless the pound (#) flag is specified.

g

Specifies a floating-point conversion. The associated argument is a floating-point number, and the result is identical to `%e` or `%f` depending on which gives the most compact result. The `%e` format is used only when the exponent is less than -4 or greater than the specified or default precision. Trailing zeroes are eliminated, and the decimal point appears only if any nonzero digits follow it.

G

Is identical to `%g`, except that `%E` gets used instead of `%e`.

hn

Is identical to `%n`, except that the argument is a pointer to a 16-bit integer instead of a 32-bit one.

i

Specifies a signed decimal integer conversion. The associated argument is a 32-bit integer, and the result is a string of digits preceded by a sign. If the plus (+) and blank flags are absent, the sign is produced only for a negative integer.

ld, li, lo, lu, lx, lX

Synonyms for `%d`, `%i`, `%o`, `%u`, `%x`, and `%X` respectively.

Ld, Li, Lo, Lu, Lx, LX

Identical to %d, %i, %o, %u, %x, and %X respectively, except that the associated argument is a 64-bit quantity instead of a 32-bit one.

Ln

Is identical to %n, except that the argument is a pointer to a 64-bit integer instead of a 32-bit one.

n, ln

Specifies the argument is a pointer to a 32-bit integer into which should be written the number of characters output so far during this formatting operation.

o

Specifies an unsigned octal integer conversion. The associated argument is an unsigned 32-bit integer, which is output in octal format.

p

Specifies a pointer conversion. The associated argument is taken as a 32-bit pointer value, and it is converted to hexadecimal. The default field precision is set to 8 for this conversion. The pound (#) flag causes a '@' symbol to be used as a prefix.

P

Is identical to %p, except that it uses uppercase letters to represent the hexadecimal digits.

s

Specifies a string conversion. The associated argument points to a character string. The string is copied to the output, up until a null byte is encountered, or the field precision is reached.

u

Specifies an unsigned decimal integer conversion. The associated argument is a 32-bit unsigned integer, which is output in decimal format.

x

Specifies an unsigned hexadecimal-integer conversion. The associated argument is an unsigned 32-bit integer, which is output in hexadecimal format using lowercase letters.

X

Is identical to %x, except that it uses uppercase letters to represent the hexadecimal digits.

Implementation Details

The following are minor extensions to the ANSI C standard, and implementation-defined behaviors for this function:

1 - The %Ld, %Li, %Lo, %Lu, %Lx, and %LX conversions types were added to deal with 64-bit integers.

2 - The %P conversion type was added. It is identical to %p, except that uppercase hexadecimal digits are used instead of lowercase ones.

3 - Unless a width is explicitly supplied, the default width for %p and %P is set to 8 hexadecimal digits.

4 - Supplying a NULL pointer to a %s conversion type will generate the output string "<NULL>".

5 - The pound (#) flags outputs a leading 0 for %o values other than 0, 0x for %x, 0X for %X, and @ for %p and %P.

6 - The IEEE floating-point indeterminate values of infinity and not-a-number are output as "Infinity" and "NaN" respectively.

Arguments

format

The format string describing the output process.

...

The optional arguments containing the data to be used during output. There must be enough arguments specified to satisfy the needs of the formatting string, otherwise strange output, or maybe even a crash, will occur.

Return Value

Returns the number of characters output or a negative error code for failure.

Associated Files

<stdio.h>, libc.a

abs
labs

Absolute Value

Synopsis

```
int abs( int i )  
long labs( long i )
```

Description

This function returns the absolute value of an integer (or a long integer).

Arguments

i
An integer (or long integer)

Return Value

Returns an integer (or long).

Implementation

Inline assembly function in `stdlib.h` and a link library function in `libc`.

Associated Files

`<stdlib.h>`, `libc.a`

Chapter 2

Batt Folio Calls

This section presents the reference documentation for the Batt folio, which provides access to the system's battery-backed clock and memory.

ReadBattClock

Reads the current setting of the battery-backed clock.

Synopsis

```
Err ReadBattClock(GregorianCalendar *gd);
```

Description

This function reads the current value of the system's battery-backed clock.

Arguments

gd

Pointer to a GregorianCalendar structure which is filled in by this function.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

BATT_ERR_NOHARDWARE

This system doesn't have any battery-backed clock.

BATT_ERR_BADPTR

The supplied buffer pointer is not valid.

Implementation

Folio call implemented in Batt folio V30.

Associated Files

<:misc:batt.h>, System.m2/Modules/batt

See Also

WriteBattClock()

WriteBattClock

Sets the battery-backed clock.

Synopsis

```
Err WriteBattClock(const GregorianCalendar *gd);
```

Description

This function sets the time and date of the system's battery-backed clock.

Arguments

gd

A GregorianCalendar structure initialized with the time and date value to set in the battery-backed clock.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

BATT_ERR_NOHARDWARE

This system doesn't have any battery-backed clock.

BATT_ERR_BADPTR

The supplied date pointer is not valid.

Implementation

Folio call implemented in Batt folio V30.

Associated Files

<:misc:batt.h>, System.m2/Modules/batt

See Also

ReadBattClock()

Chapter 3

Compression Folio Calls

This section presents the reference documentation for the Compression folio and associated link libraries.

CreateCompressor

Creates a compression engine.

Synopsis

```
Err CreateCompressor(Compressor **comp, CompFunc cf,
                    const TagArg *tags);

Err CreateCompressorVA(Compressor **comp, CompFunc cf,
                      uint32 tags, ...);
```

Description

Creates a compression engine. Once the engine is created, you can call `FeedCompressor()` to have the engine compress the data you supply.

Arguments

- comp**
A pointer to a compressor variable, where a handle to the compression engine can be put.
- cf**
A data output function. Every word of compressed data is sent to this function. This function is called with two parameters: one is a user-data value as supplied with the `COMP_TAG_USERDATA` tag. The other is the word of compressed data being output by the compressor.
- tags**
A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with `TAG_END`.

`COMP_TAG_WORKBUFFER (void *)`

A pointer to a work buffer. This buffer is used by the compressor to store state information. If this tag is not supplied, the buffer is automatically allocated and freed by the folio. To obtain the required size for the buffer, call the `GetCompressorWorkBufferSize()` function. The buffer you supply must remain valid until `DeleteCompressor()` is called. When you supply a work buffer, this routine allocates no memory of its own.

`COMP_TAG_USERDATA (void *)`

A value that the compressor will pass to `cf` when it is called. This value can be anything you want. For example, it can be a pointer to a private data structure containing some context such as a file handle. If this tag is not supplied, then `NULL` is passed to `cf` when it is called.

Return Value

Returns ≥ 0 for success, or a negative error code if the compression engine could not be created. Possible error codes include:

`COMP_ERR_BADPTR`

An invalid output function pointer or work buffer was supplied.

COMP_ERR_BADTAG

An unknown tag was supplied.

COMP_ERR_NOMEM

There was not enough memory to initialize the compressor.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

FeedCompressor(), DeleteCompressor(), GetCompressorWorkBufferSize(),
CreateDecompressor()

CreateDecompressor

Creates a decompression engine.

Synopsis

```
Err CreateDecompressor(Decompressor **comp, CompFunc cf,  
                      const TagArg *tags);  
  
Err CreateDecompressorVA(Decompressor **comp, CompFunc cf,  
                        uint32 tags, ...);
```

Description

Creates a decompression engine. Once the engine is created, you can call `FeedDecompressor()` to have the engine decompress the data you supply.

Arguments

`decomp`

A pointer to a decompressor variable, where a handle to the decompression engine can be put.

`cf`

A data output function. Every word of decompressed data is sent to this function. This function is called with two parameters: one is a user-data value as supplied with the `COMP_TAG_USERDATA` tag. The other is the word of decompressed data being output by the decompressor.

`tags`

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with `TAG_END`.

`COMP_TAG_WORKBUFFER` (void *)

A pointer to a work buffer. This buffer is used by the decompressor to store state information. If this tag is not supplied, the buffer is automatically allocated and freed by the folio. To obtain the required size for the buffer, call the `GetDecompressorWorkBufferSize()` function. The buffer you supply must remain valid until `DeleteDecompressor()` is called. When you supply a work buffer, this routine allocates no memory of its own.

`COMP_TAG_USERDATA` (void *)

A value that the decompressor will pass to `cf` when it is called. This value can be anything you want. For example, it can be a pointer to a private data structure containing some context such as a file handle. If this tag is not supplied, then `NULL` is passed to `cf` when it is called.

Return Value

Returns ≥ 0 for success, or a negative error code if the decompression engine could not be created. Possible error codes include:

`COMP_ERR_BADPTR`

An invalid output function pointer or work buffer was supplied.

COMP_ERR_BADTAG

An unknown tag was supplied.

COMP_ERR_NOMEM

There was not enough memory to initialize the decompressor.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

FeedDecompressor(), DeleteDecompressor(),
GetDecompressorWorkBufferSize(), CreateCompressor()

DeleteCompressor

Deletes a compression engine.

Synopsis

```
Err DeleteCompressor(Compressor *comp);
```

Description

Deletes a compression engine previously created by `CreateCompressor()`. This flushes any data left to be output by the compressor and generally cleans things up.

Arguments

`comp`

An active compression handle, as obtained from `CreateCompressor()`. Once this call is made, the compression handle becomes invalid and can no longer be used.

Return Value

Returns ≥ 0 if successful, or a negative error code if it fails. Possible error codes include:

`COMP_ERR_BADPTR`

An invalid compression handle was supplied.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

`<:misc:compression.h>`, `System.m2/Modules/compression`

See Also

`CreateCompressor()`

DeleteDecompressor

Deletes a decompression engine.

Synopsis

```
Err DeleteDecompressor(Decompressor *decomp);
```

Description

Deletes a decompression engine previously created by `CreateDecompressor()`. This flushes any data left to be output by the decompressor and generally cleans things up.

Arguments

`decomp`

An active decompression handle, as obtained from `CreateDecompressor()`. Once this call is made, the decompression handle becomes invalid and can no longer be used.

Return Value

Returns ≥ 0 for success, or a negative error code if it fails. Possible error codes include:

`COMP_ERR_BADPTR`

An invalid decompression handle was supplied.

`COMP_ERR_DATAREMAINS`

The decompressor thinks it is finished, but there remains extra data in its buffers. This happens when the compressed data is somehow corrupt.

`COMP_ERR_DATAMISSING`

The decompressor thinks that not all of the compressed data was given to be decompressed. This happens when the compressed data is somehow corrupt.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

`<:misc:compression.h>`, `System.m2/Modules/compression`

See Also

`CreateCompressor()`

FeedCompressor

Gives data to a compression engine.

Synopsis

```
Err FeedCompressor(Compressor *comp, const void *data,  
                  uint32 numDataWords);
```

Description

Gives data to a compressor engine for compression. As data is compressed, the call back function supplied when the compressor was created is called for every word of compressed data generated.

Arguments

- comp
An active compression handle, as obtained from `CreateCompressor()`.
- data
A pointer to the data to compress.
- numDataWords
The number of words of data being given to the compressor.

Return Value

Returns ≥ 0 for success, or a negative error code if it fails. Possible error codes include:

- COMP_ERR_BADPTR
An invalid compression handle was supplied.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

`<:misc:compression.h>`, `System.m2/Modules/compression`

See Also

`CreateCompressor()`, `DeleteCompressor()`

FeedDecompressor

Gives data to a decompression engine.

Synopsis

```
Err FeedDecompressor(Decompressor *decomp, const void *data,  
                     uint32 numDataWords);
```

Description

Gives data to the decompressor engine for decompression. As data is decompressed, the call back function supplied when the decompressor was created is called for every word of decompressed data generated.

Arguments

decomp

An active decompression handle, as obtained from `CreateDecompressor()`.

data

A pointer to the data to decompress.

numDataWords

The number of words of compressed data being given to the decompressor.

Return Value

Returns ≥ 0 for success, or a negative error code if it fail. Possible error codes include:

COMP_ERR_BADPTR

An invalid decompression handle was supplied.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

`CreateDecompressor()`, `DeleteDecompressor()`

GetCompressorWorkBufferSize

Gets the size of the work buffer needed by a compression engine.

Synopsis

```
int32 GetCompressorWorkBufferSize(const TagArg *tags);  
  
int32 GetCompressorWorkBufferSizeVA(uint32 tags, ...);
```

Description

Returns the size of the work buffer needed by a compression engine. You can then allocate a buffer of that size and supply the pointer with the COMP_TAG_WORKBUFFER tag when creating a compression engine. If the COMP_TAG_WORKBUFFER tag is not supplied when creating a compressor, the folio automatically allocates the memory needed for the compression engine.

Arguments

tags
A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

A positive value indicates the size of the work buffer needed in bytes, while a negative value indicates an error. Possible error codes currently include:

COMP_ERR_BADTAG
An unknown tag was supplied.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

CreateCompressor(), DeleteCompressor()

GetDecompressorWorkBufferSize Gets the size of the work buffer needed by a decompression engine.

Synopsis

```
int32 GetDecompressorWorkBufferSize(const TagArg *tags);  
  
int32 GetDecompressorWorkBufferSizeVA(uint32 tags, ...);
```

Description

Returns the size of the work buffer needed by a decompression engine. You can then allocate a buffer of that size and supply the pointer with the COMP_TAG_WORKBUFFER tag when creating a decompression engine. If the COMP_TAG_WORKBUFFER tag is not supplied when creating a decompressor, the folio automatically allocates the memory needed for the decompression engine.

Arguments

tags
A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

A positive value indicates the size of the work buffer needed in bytes, while a negative value indicates an error. Possible error codes currently include:

COMP_ERR_BADTAG
An unknown tag was supplied.

Implementation

Folio call implemented in Compression folio V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

CreateDecompressor(), DeleteDecompressor()

SimpleCompress

Compresses some data in memory.

Synopsis

```
Err SimpleCompress(const void *source, uint32 sourceWords,  
                  void *result, uint32 resultWords);
```

Description

Compresses a chunk of memory to a different chunk of memory.

Arguments

source

A pointer to memory containing the data to be compressed.

sourceWords

The number of words of data to compress.

result

A pointer to where the compressed data is to be deposited.

resultWords

The number of words available in the result buffer. If the compressed data is larger than this size, an overflow will be reported.

Return Value

If the return value is positive, it indicates the number of words copied to the result buffer. If the return value is negative, it indicates an error code. Possible error codes include:

COMP_ERR_NOMEM

There was not enough memory to initialize the compressor.

COMP_ERR_OVERFLOW

There was not enough room in the result buffer to hold all of the compressed data.

Implementation

Convenience call implemented in System.m2/Modules/compression V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

SimpleDecompress()

SimpleDecompress

Decompresses some data in memory.

Synopsis

```
Err SimpleDecompress(const void *source, uint32 sourceWords,  
                    void *result, uint32 resultWords);
```

Description

Decompresses a chunk of memory to a different chunk of memory.

Arguments

source

A pointer to memory containing the data to be decompressed.

sourceWords

The number of words of data to decompress.

result

A pointer to where the decompressed data is to be deposited.

resultWords

The number of words available in the result buffer. If the decompressed data is larger than this size, an overflow will be reported.

Return Value

If positive, returns the number of words copied to the result buffer. If negative, returns an error code. Possible error codes include:

COMP_ERR_NOMEM

There was not enough memory to initialize the decompressor.

COMP_ERR_OVERFLOW

There was not enough room in the result buffer to hold all of the decompressed data.

Implementation

Convenience call implemented in System.m2/Modules/compression V24.

Associated Files

<:misc:compression.h>, System.m2/Modules/compression

See Also

SimpleCompress()

Chapter 4

Date Folio Calls

This section presents the reference documentation for the Date folio, which provides conversion functions to handle date and time values.

ConvertGregorianToTimeVal

Converts from a real-world date representation to a system TimeVal.

Synopsis

```
Err ConvertGregorianToTimeVal(const GregorianDate *gd, TimeVal *tv);
```

Description

This function converts from a gregorian date format, which is expressed in terms of year, month, and day, into a system TimeVal format, which specifies a date in number of seconds from January 1st 1993.

Arguments

gd

The GregorianDate to convert.

tv

A TimeVal structure where the converted date is put.

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

Folio call implemented in Date folio V30.

Associated Files

<:misc:date.h>, System.m2/Modules/date

See Also

ConvertTimeValToGregorian()

ConvertTimeValToGregorian

Converts from a system TimeVal to a real-world date representation.

Synopsis

```
Err ConvertTimeValToGregorian(const TimeVal *tv, GregorianDate *gd);
```

Description

This function converts from a TimeVal format, which specifies a date in number of seconds from January 1st 1993, to a gregorian date format which is expressed in terms of year, month, and day.

Arguments

tv
The TimeVal to convert

gd
A GregorianDate structure where the converted value is put.

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

Folio call implemented in Date folio V30.

Associated Files

<:misc:date.h>, System.m2/Modules/date

See Also

ConvertGregorianToTimeVal()

ValidateDate

Makes sure a date is valid.

Synopsis

```
Err ValidateDate(const GregorianCalendar *gd);
```

Description

This function makes sure that the supplied date is valid and doesn't contain out-of-bounds values.

Arguments

gd
Pointer to an initialized GregorianCalendar structure to validate.

Return Value

Returns ≥ 0 if the date is valid, or a negative error code if it isn't.

Implementation

Folio call implemented in Date folio V30.

Associated Files

<:misc:date.h>, System.m2/Modules/date

Chapter 5

Debug Console Link Library Calls

This section presents the reference documentation for the debug console link library, which provides services to perform console-style output to a 3DO View.

CreateDebugConsole

Creates a view for debugging console output.

Synopsis

```
Err CreateDebugConsole(const TagArg *tags);  
Err CreateDebugConsoleVA(uint32 tag, ...);
```

Description

Outputting text to the standard debugging terminal using `printf()` is a fairly slow proposal, involving a fair amount of overhead and delays, making it hard to use `printf()` in time-sensitive code.

This function creates a view on the front of the 3DO display where debugging output can be sent. This is a more efficient and less intrusive way to perform debugging output. You use the `DebugConsolePrintf()` call to output the text instead of `printf()`.

The various routines that affect the debugging console are protected to allow multiple threads to write to the debug output simultaneously and not interfere with one another.

Arguments

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with `TAG_END`.

`DEBUGCONSOLE_TAG_HEIGHT` (uint32)

This tag specifies the height in pixels of the debugging view console. Defaults to 200.

`DEBUGCONSOLE_TAG_TOP` (uint32)

This tag specifies the position of the top of the debugging view console relative to the top of the entire display. Defaults to 0.

`DEBUGCONSOLE_TAG_TYPE` (enum ViewType)

This tag specifies the type of view to use for the debugging console. See [<:graphics:view.h>](#) for a definition of the possible view types. This defaults to `VIEWTYPE_16_640_LACE`.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Link library call implemented in `libdebugconsole.a` V27.

Associated Files

[<:misc:debugconsole.h>](#), `libdebugconsole.a`

See Also

`DeleteDebugConsole()`, `DebugConsolePrintf()`, `DebugConsoleClear()`,
`DebugConsoleMove()`

DebugConsoleClear

Erases everything in a debugging view console.

Synopsis

```
void DebugConsoleClear(void);
```

Description

This function erases everything in the debugging view console and resets everything to the background color.

Implementation

Link library call implemented in libdebugconsole.a V27.

Associated Files

<:misc:debugconsole.h>, libdebugconsole.a

See Also

CreateDebugConsole(), DeleteDebugConsole(), DebugConsolePrintf()
DebugConsoleMove()

DebugConsoleMove

Moves the debugging console rendering cursor.

Synopsis

```
void DebugConsoleMove(uint32 x, uint32 y);
```

Description

This function lets you move the cursor which determines where the next character will be displayed when printing to the debugging view console.

The coordinates are relative to the top left corner of the view console and are specified in pixels.

Arguments

x
The new horizontal pixel position of the cursor.

y
The new vertical pixel position of the cursor.

Implementation

Link library call implemented in libdebugconsole.a V27.

Associated Files

<:misc:debugconsole.h>, libdebugconsole.a

See Also

CreateDebugConsole(), DeleteDebugConsole(), DebugConsolePrintf()
DebugConsoleClear(), DebugConsoleMove()

DebugConsolePrintf

Output information to a debugging view console.

Synopsis

```
void DebugConsolePrintf(const char *text, ...);
```

Description

This function works like `printf()`, with the difference that the output is sent to a debugging view console on the 3DO display instead of to the debugging terminal.

Arguments

Same as `printf()`

Implementation

Link library call implemented in `libdebugconsole.a V27`.

Associated Files

<*misc:debugconsole.h*>, `libdebugconsole.a`

See Also

`CreateDebugConsole()`, `DeleteDebugConsole()`, `DebugConsoleClear()`,
`DebugConsoleMove()`

DeleteDebugConsole

Closes the debugging console view preventing further debugging output.

Synopsis

```
void DeleteDebugConsole(void);
```

Description

This function takes down the debugging console, and releases any resources allocated by `CreateDebugConsole()`.

Implementation

Link library call implemented in `libdebugconsole.a V27`.

Associated Files

<*misc:debugconsole.h*>, `libdebugconsole.a`

See Also

`CreateDebugConsole()`, `DebugConsolePrintf()` `DebugConsoleClear()`,
`DebugConsoleMove()`

Chapter 6

Device Command Reference

This section presents the reference documentation for the many I/O commands supported by the various devices in the Portfolio architecture.

CDROMCMD_CLOSE_DRAWER Close the CD-ROM drive's drawer.**Description**

This command closes the CD-ROM drive drawer. This will not have any effect on a clamshell mechanism.

IOInfo

ioi_Command

Set to CDROMCMD_CLOSE_DRAWER.

Implementation

Command defined in Portfolio V27.

Associated Files

<:device:cdrom.h>, <:kernel:devicecmd.h>

See Also

CDROMCMD_OPEN_DRAWER

CDROMCMD_DISCDATA

Requests the Disc Information (DiscID, TOC, and Session Info).

Description

This command causes the driver to return the DiscID, Table Of Contents, and Session Info for the current disc.

IOInfo

`ioi_Command`

Set to `CDROMCMD_DISCDATA`.

`ioi_Recv.iob_Buffer`

Pointer to a `CDROM_Disc_Data` structure.

`ioi_Recv.iob_Len`

Set to `sizeof(CDROM_Disc_Data)`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:device:cdrom.h>`, `<:kernel:devicecmd.h>`

CDROMCMD_OPEN_DRAWER Open the CD-ROM drive's drawer.**Description**

This command opens the CD-ROM drive drawer. This will not have any effect on a clamshell mechanism.

IOInfo

ioi_Command

Set to CDROMCMD_OPEN_DRAWER.

Implementation

Command defined in Portfolio V27.

Associated Files

<:device:cdrom.h>, <:kernel:devicecmd.h>

See Also

CDROMCMD_CLOSE_DRAWER

CDROMCMD_READ

Requests specific block (sector) data from the disc.

Description

This command causes the requested sector(s) to be returned. This command is similar to `CMD_BLOCKREAD`; but allows various options to be supplied in the `ioi_CmdOptions` field. The block size is generally defined by the media present.

IOInfo`ioi_Command`Set to `CDROMCMD_READ`.`ioi_Recv.iob_Buffer`

Pointer to a receive buffer. Optimum performance will be attained if the buffer is 4-byte aligned.

`ioi_Recv.iob_Len`The length of the receive buffer must be a multiple of the block size. Again, this is dependent on the current media. For the pre-defined block lengths, see `ioi_CmdOptions.asFields.blockLength` below.`ioi_Offset`

This is the starting address of the sector(s) you wish to read. It can be in either of the following formats: 1) the absolute sector address (e.g., sector 472903); or 2) the address of the sector in binary MSF format.

`ioi_CmdOptions`This field provides access to parameters which define how to perform this read. The options are provided as bitfields of this 32-bit word. Specifying a value of zero for each field means you wish to use the current default. The system defaults are indicated below with '*'. The fields are accessed by specifying '`ioi_CmdOptions.asFields`' before the following:`.densityCode`

Specifies what type of media you are reading; and is one of the following:

NULL

Use the current default.

`CDROM_DEFAULT_DENSITY*`

Indicates that you wish to read Mode1, Mode2Form1, or Mode2Form2 type sectors. Do whatever is necessary to return the correct data.

`CDROM_DATA`

Indicates that you wish to read Mode1 sectors. If a different sector type exists an error is returned.

`CDROM_MODE2_XA`

Indicates that you wish to read Mode2Form1 or Mode2Form2 sectors. If a different sector type exists an error is returned.

`CDROM_DIGITAL_AUDIO`

Indicates that you wish to read RedBook Audio sectors data.

.addressFormat

Indicates what address format is specified in `ioi_Offset`. Valid values are:

NULL

Use the current default.

CDROM_Address_Blocks*

Indicates that the sector address specified in `ioi_Offset` is an absolute block/sector number.

CDROM_Address_Abs_MSF

Indicates that the sector address specified in `ioi_Offset` is specified in binary MSF format. That is, a 32-bit value (0x00MMSSFF) in which each field (M,S,F) is binary (as opposed to Binary Coded Decimal).

.errorRecovery

Specifies what type of error recovery you would like performed; and whether or not any errored sector data is returned. Valid values are:

NULL

Use the current default.

CDROM_DEFAULT_RECOVERY*

Perform ECC on any errored sector. If ECC fails, retry the read. Continue for the number of retries specified by `.retryShift`. If unable to obtain the sector cleanly, return an error. NOTE: Does not return the errored sector data.

CDROM_CIRC_RETRIES_ONLY

Perform retries only (no ECC). If retries fail, return an error. NOTE: This does return the errored sector data.

CDROM_BEST_ATTEMPT_RECOVERY

Perform ECC and retries. If both fail, return an error AND the errored sector. (Similar to `CDROM_DEFAULT_RECOVERY`.)

.retryShift

Specifies the number of retries that the driver attempts for a given sector when an error is detected. Upon exceeding this count, the driver returns an error. If the `.errorRecovery` field contained `CDROM_BEST_ATTEMPT_RECOVERY`, then the errored sector data is returned as well.

The value (n) specified indicates that you wish to perform $(2^n - 1)$ retries. Valid values for n range from 0 (zero retries) to 7 (127 retries). The default value is 3 (7 retries).

.speed

Specifies the speed at which the data is to be read. Some mechanisms may support 4x, 6x,

and 8x speeds. Others return an error. Valid values are:

NULL

Use the current default.

CDROM_SINGLE_SPEED

Operate the mechanism at single speed.

CDROM_DOUBLE_SPEED*

Operate the mechanism at double speed.

CDROM_4X_SPEED

Operate the mechanism at 4x speed.

CDROM_6X_SPEED

Operate the mechanism at 6x speed.

CDROM_8X_SPEED

Operate the mechanism at 8x speed.

.pitch

Specifies the variable pitch component of the speed. Note that this is only valid when operating in CDROM_SINGLE_SPEED. Also note that some mechanisms may not contain the hardware to support variable pitch. In such a case NO error is returned; and the data is read in normal pitch. Valid values are:

NULL

Use the current default.

CDROM_PITCH_SLOW

Operate drive at -1% of single speed.

CDROM_PITCH_NORMAL

Operate drive at single speed.

CDROM_PITCH_FAST

Operate drive at +1% of single speed.

.blockLength

Specifies the block length that you are interested in reading from the disc sector. Note that this value is dependent upon the media present. The pre-defined values are:

NULL

Use the current default.

CDROM_AUDIO

Returns 2352 bytes of audio data for each sector.

CDROM_AUDIO_SUBCODE

Returns 2352 bytes of (RedBook) Audio data, followed by 96 bytes of Subcode, for EACH sector. NOTE: The subcode returned with each sector is only LOOSELY associated with the

sector returned. The subcode is NOT synced up with the data for each sector.

CDROM_MODE1*

Returns 2048 bytes of Mode1 (YellowBook) sector data.

CDROM_MODE2FORM1

Returns 2048 bytes of Mode2Form1 (OrangeBook) sector data.

CDROM_MODE2FORM1_SUBHEADER

Returns 8 bytes of SubHeader, followed by 2048 bytes of Mode2Form1 (OrangeBook) sector data for EACH sector.

CDROM_MODE2FORM2

Returns 2324 bytes of Mode2Form2 (OrangeBook) sector data.

CDROM_MODE2FORM2_SUBHEADER

Returns 8 bytes of SubHeader, followed by 2324 bytes of Mode2Form2 (OrangeBook) sector data for EACH sector.

Note that other valid block lengths are possible. These would rarely be used; and it is left as an experiment to the reader to determine what they are. A hint: any combination of Header + SubHeader + Data + Aux/ECC can generally be obtained per sector.

Implementation

Command defined in Portfolio V27.

Associated Files

<:device:cdrom.h>, <:kernel:devicecmd.h>

Notes

Much information about the contents of sector data can be gleaned from the RedBook, YellowBook, OrangeBook, WhiteBook, and GreenBook CD-ROM specifications (Phillips/Sony).

See Also

CDROMCMD_SCAN_READ, CDROMCMD_SETDEFAULTS, CMD_BLOCKREAD

CDROMCMD_READ_SUBQ

Requests the QCode data for the current sector.

Description

This command causes the driver to return the QCode information for the current sector.

IOInfo

ioi_Command

Set to CDROMCMD_READ_SUBQ.

ioi_Recv.iob_Buffer

Pointer to a SubQInfo structure.

ioi_Recv.iob_Len

Set to sizeof(SubQInfo).

Implementation

Command defined in Portfolio V27.

Associated Files

<:device:cdrom.h>, <:kernel:devicecmd.h>

CDROMCMD_SCAN_READ

Requests approximate block (sector) data from the disc.

Description

This command causes the drive to seek "close to" the requested sector and begin returning data immediately. This provides a means of loose seeking by allowing (cd-rom) track skipping without having to return a specific sector. It is anticipated that this will be used by the Audio and Video apps to improve fast-fwd/fast-rev performance.

IOInfo**ioi_Command**

Set to CDROMCMD_SCAN_READ.

ioi_Recv.iob_Buffer

Pointer to a receive buffer. Optimum performance will be attained if the buffer is 4-byte aligned.

ioi_Recv.iob_Len

The length of the receive buffer must be a multiple of the block size. This is dependent on the current media. For the pre-defined block lengths, see `ioi_CmdOptions.asFields.blockLength` below.

ioi_Offset

This is the address of the sector you wish to jump close to. It can be in either of the following formats: 1) the absolute sector address (e.g., sector 9385681); or 2) the address of the sector in binary MSF format.

ioi_CmdOptions

This field provides access to parameters which define how to perform this read. The options are provided as bitfields of this 32-bit word. Specifying a value of zero for each field means you wish to use the current default. The system defaults are indicated below with '*'. The fields are accessed by specifying 'ioi_CmdOptions.asFields' before the following:

.densityCode

Specifies what type of media you are reading; and is one of the following:

NULL

Use the current default.

CDROM_DEFAULT_DENSITY*

Indicates that you wish to read Mode1, Mode2Form1, or Mode2Form2 type sectors. Do whatever is necessary to return the correct data.

CDROM_DATA

Indicates that you wish to read Mode1 sectors. If a different sector type exists an error is returned.

CDROM_MODE2_XA

Indicates that you wish to read Mode2Form1 or Mode2Form2 sectors. If a different sector type exists an error is returned.

CDROM_DIGITAL_AUDIO

Indicates that you wish to read RedBook Audio sectors data.

.addressFormat

Indicates what address format is specified in `ioi_Offset`. Valid values are:

NULL

Use the current default.

CDROM_Address_Blocks*

Indicates that the sector address specified in `ioi_Offset` is an absolute block/sector number.

CDROM_Address_Abs_MSF

Indicates that the sector address specified in `ioi_Offset` is specified in binary MSF format. That is, a 32-bit value (0x00MMSSFF) in which each field (M,S,F) is binary (as opposed to Binary Coded Decimal).

.errorRecovery

Specifies what type of error recovery you would like performed; and whether or not any errored sector data is returned. Valid values are:

NULL

Use the current default.

CDROM_DEFAULT_RECOVERY*

Perform ECC on any errored sector. If ECC fails, retry the read. Continue for the number of retries specified by `.retryShift`. If unable to obtain the sector cleanly, return an error. NOTE: Does not return the errored sector data.

CDROM_CIRC_RETRIES_ONLY

Perform retries only (no ECC). If retries fail, return an error. NOTE: Does not return the errored sector data.

CDROM_BEST_ATTEMPT_RECOVERY

Perform ECC and retries. If both fail, return an error AND the errored sector. (Similar to `CDROM_DEFAULT_RECOVERY`.)

.retryShift

Specifies the number of retries that the driver attempts for a given sector when an error is detected. Upon exceeding this count, the driver returns an error. If the `.errorRecovery` field contained `CDROM_BEST_ATTEMPT_RECOVERY`, then the errored sector data is returned as well.

The value (n) specified indicates that you wish to perform $(2^n - 1)$ retries. Valid values for n range from 0 (zero retries) to 7 (127 retries). The default value is 3 (7 retries).

.speed

Specifies the speed at which the data is to be read. Some mechanisms may support 4x, 6x, and 8x speeds. Others return an error. Valid values are:

NULL

Use the current default.

CDROM_SINGLE_SPEED

Operate the mechanism at single speed.

CDROM_DOUBLE_SPEED*

Operate the mechanism at double speed.

CDROM_4X_SPEED

Operate the mechanism at 4x speed.

CDROM_6X_SPEED

Operate the mechanism at 6x speed.

CDROM_8X_SPEED

Operate the mechanism at 8x speed.

.pitch

Specifies the variable pitch component of the speed. Note that this is only valid when operating in CDROM_SINGLE_SPEED. Also note that some mechanisms may not contain the hardware to support variable pitch. In such a case NO error is returned; and the data is read in normal pitch. Valid values are:

NULL

Use the current default.

CDROM_PITCH_SLOW

Operate drive at -1% of single speed.

CDROM_PITCH_NORMAL

Operate drive at single speed.

CDROM_PITCH_FAST

Operate drive at +1% of single speed.

.blockLength

Specifies the block length that you are interested in reading from the disc sector. Note that this value is dependent upon the media present. The pre-defined values are:

NULL

Use the current default.

CDROM_AUDIO

Returns 2352 bytes of audio data for each sector.

CDROM_AUDIO_SUBCODE

Returns 2352 bytes of (RedBook) Audio data, followed by 96 bytes of Subcode, for EACH sector. NOTE: The subcode

returned with each sector is only LOOSELY associated with the sector returned. The subcode is NOT synced up with the data for each sector.

CDROM_MODE1*

Returns 2048 bytes of Mode1 (YellowBook) sector data.

CDROM_MODE2FORM1

Returns 2048 bytes of Mode2Form1 (OrangeBook) sector data.

CDROM_MODE2FORM1_SUBHEADER

Returns 8 bytes of SubHeader, followed by 2048 bytes of Mode2Form1 (OrangeBook) sector data for EACH sector.

CDROM_MODE2FORM2

Returns 2324 bytes of Mode2Form2 (OrangeBook) sector data.

CDROM_MODE2FORM2_SUBHEADER

Returns 8 bytes of SubHeader, followed by 2324 bytes of Mode2Form2 (OrangeBook) sector data for EACH sector.

Note that other valid block lengths are possible. These would rarely be used; and it is left as an experiment to the reader to determine what they are. A hint: any combination of Header + SubHeader + Data + Aux/ECC can generally be obtained per sector.

Implementation

Command defined in Portfolio V27.

Associated Files

<:device:cdrom.h>, <:kernel:devicecmd.h>

Notes

Much information about the contents of sector data can be gleaned from the RedBook, YellowBook, OrangeBook, WhiteBook, and GreenBook CD-ROM specifications (Phillips/Sony).

See Also

CDROMCMD_READ, CDROMCMD_SETDEFAULTS, CMD_BLOCKREAD

CDROMCMD_SETDEFAULTS

Set/Get the current device defaults for read-mode IOReqs.

Description

This command sets the internal defaults used by the CD-ROM driver to process CDROMCMD_READ and CDROMCMD_SCAN_READ IOReqs. Each of these defaults is only used if its associated field in ioi_CmdOptions is set to NULL.

IOInfo

ioi_Command

Set to CDROMCMD_READ.

ioi_Recv.iob_Buffer

To obtain the current defaults, set this to the address of a CDROMCommandOptions structure.

ioi_Recv.iob_Len

If obtaining the current defaults, set to sizeof(CDROMCommandOptions).

ioi_CmdOptions

This field provides is where the values for the new defaults are specified. The options are provided as bitfields of this 32-bit word. Specifying a value of zero for each field means you wish to keep the current default. The system defaults are indicated below with '*'. The fields are accessed by specifying 'ioi_CmdOptions.asFields' before the following:

.densityCode

Specifies what type of media you are reading; and is one of the following:

NULL

Use the current default.

CDROM_DEFAULT_DENSITY*

Indicates that you wish to read Mode1, Mode2Form1, or Mode2Form2 type sectors. Do whatever is necessary to return the correct data.

CDROM_DATA

Indicates that you wish to read Mode1 sectors. If a different sector type exists an error is returned.

CDROM_MODE2_XA

Indicates that you wish to read Mode2Form1 or Mode2Form2 sectors. If a different sector type exists an error is returned.

CDROM_DIGITAL_AUDIO

Indicates that you wish to read RedBook Audio sectors data.

.addressFormat

Indicates what address format is specified in ioi_Offset. Valid values are:

NULL

Use the current default.

CDROM_Address_Blocks*

Indicates that the sector address specified in `ioi_Offset` is an absolute block/sector number.

CDROM_Address_Abs_MSF

Indicates that the sector address specified in `ioi_Offset` is specified in binary MSF format. That is, a 32-bit value (0x00MMSSFF) in which each field (M,S,F) is binary (as opposed to Binary Coded Decimal).

.errorRecovery

Specifies what type of error recovery you would like performed; and whether or not any errored sector data is returned. Valid values are:

NULL

Use the current default.

CDROM_DEFAULT_RECOVERY*

Perform ECC on any errored sector. If ECC fails, retry the read. Continue for the number of retries specified by `.retryShift`. If unable to obtain the sector cleanly, return an error. NOTE: Does not return the errored sector data.

CDROM_CIRC_RETRIES_ONLY

Perform retries only (no ECC). If retries fail, return an error. NOTE: Does not return the errored sector data.

CDROM_BEST_ATTEMPT_RECOVERY

Perform ECC and retries. If both fail, return an error AND the errored sector. (Similar to `CDROM_DEFAULT_RECOVERY`.)

.retryShift

Specifies the number of retries that the driver attempts for a given sector when an error is detected. Upon exceeding this count, the driver returns an error. If the `.errorRecovery` field contained `CDROM_BEST_ATTEMPT_RECOVERY`, then the errored sector data is returned as well.

The value (n) specified indicates that you wish to perform $(2^n - 1)$ retries. Valid values for n range from 0 (zero retries) to 7 (127 retries). The default value is 3 (7 retries).

Note that the only way to specify ZERO retries is if the `.errorRecovery` field is non-NULL.

.speed

Specifies the speed at which the data is to be read. Some mechanisms may support 4x, 6x, and 8x speeds. Others return an error. Valid values are:

NULL

Use the current default.

CDROM_SINGLE_SPEED

Operate the mechanism at single speed.

CDROM_DOUBLE_SPEED*

Operate the mechanism at double speed.

CDROM_4X_SPEED

Operate the mechanism at 4x speed.

CDROM_6X_SPEED

Operate the mechanism at 6x speed.

CDROM_8X_SPEED

Operate the mechanism at 8x speed.

.pitch

Specifies the variable pitch component of the speed. Note that this is only valid when operating in **CDROM_SINGLE_SPEED**. Also note that some mechanisms may not contain the hardware to support variable pitch. In such a case NO error is returned; and the data is read in normal pitch. Valid values are:

NULL

Use the current default.

CDROM_PITCH_SLOW

Operate drive at -1% of single speed.

CDROM_PITCH_NORMAL

Operate drive at single speed.

CDROM_PITCH_FAST

Operate drive at +1% of single speed.

.blockLength

Specifies the block length that you are interested in reading from the disc sector. Note that this value is dependent upon the media present. The pre-defined values are:

NULL

Use the current default.

CDROM_AUDIO

Returns 2352 bytes of audio data for each sector.

CDROM_AUDIO_SUBCODE

Returns 2352 bytes of (RedBook) Audio data, followed by 96 bytes of Subcode, for EACH sector. NOTE: The subcode returned with each sector is only LOOSELY associated with the sector returned. The subcode is NOT synced up with the data for each sector.

CDROM_MODE1*

Returns 2048 bytes of Mode1 (YellowBook) sector data.

CDROM_MODE2FORM1

Returns 2048 bytes of Mode2Form1 (OrangeBook) sector data.

CDROM_MODE2FORM1_SUBHEADER

Returns 8 bytes of SubHeader, followed by 2048 bytes of Mode2Form1 (OrangeBook) sector data for EACH sector.

CDROM_MODE2FORM2

Returns 2324 bytes of Mode2Form2 (OrangeBook) sector data.

CDROM_MODE2FORM2_SUBHEADER

Returns 8 bytes of SubHeader, followed by 2324 bytes of Mode2Form2 (OrangeBook) sector data for EACH sector.

Note that other valid block lengths are possible. These would rarely be used; and it is left as an experiment to the reader to determine what they are. A hint: any combination of Header + SubHeader + Data + Aux/ECC can generally be obtained per sector.

Implementation

Command defined in Portfolio V27.

Associated Files

<:device:cdrom.h>, *<:kernel:devicecmd.h>*

See Also

CDROMCMD_READ, CDROMCMD_SCAN_READ, CMD_BLOCKREAD

FILECMD_ALLOCBLOCKS

Allocate storage space for a file

Description

This command allocates the specified number of blocks to an existing file. Target File must have been previously created through `CreateFile()` and opened via `OpenFile()`. Space allocation is atomic, either the specified number of blocks is allocated or non. If there is not enough space on the filesystem to service the request, then an error is returned in `io_Error` field of `ioReq`. Otherwise, zero is returned. Although the filesystem makes every effort to make the allocation contiguous, caller should make no assumption about the nature of allocation. Space allocation may or may not be contiguous. This command may be called multiple times to allocate more storage space for a given file. However, it is most efficient to allocate all necessary storage only once immediately after the file is created and opened.

IOInfo

`ioi_Command`

Set to `FILECMD_ALLOCBLOCKS`

`ioi_Offset`

Number of file blocks to allocate or free.

Implementation

Command defined in Portfolio V27.

Associated Files

<:file:filesystem.h>, <:kernel:devicecmd.h>

See Also

`FILECMD_READENTRY`, `FILECMD_READDIR`, `FILECMD_SETVERSION`, `FILECMD_GETPATH`, `FILECMD_SETEOF`, `FILECMD_ADDENTRY`, `FILECMD_DELETEENTRY`, `FILECMD_SETTYPE`, `FILECMD_FSSTAT`, `FILECMD_ADDDIR`, `FILECMD_DELETEDIR`, `OpenFile()`, `CloseFile()`, `CreateFile()`

FILECMD_FSSTAT

Get filesystem status information

Description

This command acquires information about a mounted filesystem. This command can be called on any open file belonging to the target filesystem. Typical information provided by this command is the block size of the filesystem, its size, device information, and the mount point of the filesystem. Not all filesystems can provide all this information. For details see `FileSystemStat` struct in `filesystem.h`. The `fst_Bitmap` field indicates about which other fields of the structure this command was able to get information. Since not all fields are meaningful for all of the filesystems currently available on Portfolio, Target file must have been previously opened through `OpenFile()`.

IOInfo`ioi_Command`Set to `FILECMD_FSSTAT``ioi_Recv.iob_Buffer`Pointer to the user allocated `FileSystemStat` structure. After the command is successfully completed, this buffer contains the acquired data.`ioi_Recv.iob_Len`Set to `sizeof(FileSystemStat)`.**Implementation**

Command defined in Portfolio V27.

Associated Files

<:file:filesystem.h>, <:kernel:devicecmd.h>

See Also

`FILECMD_READENTRY`, `FILECMD_READDIR`, `FILECMD_ALLOCBLOCKS`,
`FILECMD_GETPATH`, `FILECMD_SETVERSION`, `FILECMD_ADDENTRY`,
`FILECMD_DELETEENTRY`, `FILECMD_SETEOF`, `FILECMD_SETTYPE`, `FILECMD_ADDDIR`,
`FILECMD_DELETEDIR`, `OpenFile()`, `CloseFile()`

FILECMD_GETPATH

Get pathname of a file

Description

This command returns the pathname of an open file. This is useful when the file has been opened by an alternate pathname. Target file must have been previously opened through `OpenFile()`.

IOInfo

`ioi_Command`

Set to `FILECMD_GETPATH`

`ioi_Recv.iob_Buffer`

Pointer to a buffer of at least `FILESYSTEM_MAX_PATH_LEN` bytes.

`ioi_Recv.iob_Len`

length of the buffer.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:file:filesystem.h>`, `<:kernel:devicecmd.h>`

See Also

`FILECMD_READENTRY`, `FILECMD_READDIR`, `FILECMD_SETVERSION`,
`FILECMD_ALLOCBLOCKS`, `FILECMD_SETEOF`, `FILECMD_ADDENTRY`,
`FILECMD_DELETEENTRY`, `FILECMD_SETTYPE`, `FILECMD_FSSTAT`, `FILECMD_ADDDIR`,
`FILECMD_DELETEDIR`, `OpenFile()`, `CloseFile()`

FILECMD_READDIR

Read a directory entry by number

Description

This command allows the caller to scan through a list of directory entries in the target directory. The number of a directory entry is its physical position in the directory block of the target directory. A directory entry is either a file or a subdirectory. Target directory must have been previously opened through `OpenFile()`.

IOInfo`ioi_Command`Set to `FILECMD_READDIR``ioi_Offset`

Number of the directory entry in the directory list. Entry 1 is the first entry in the directory.

`ioi_Recv.iob_Buffer`Pointer to the user allocated `DirectoryEntry` structure. After the command is successfully completed, this buffer contains the entry data.`ioi_Recv.iob_Len`Set to `sizeof(DirectoryEntry)`.**Implementation**

Command defined in Portfolio V27.

Associated Files

<:file:filesystem.h>, <:file:directory.h>, <:kernel:devicecmd.h>

See Also

`FILECMD_GETPATH`, `FILECMD_READENTRY`, `FILECMD_SETVERSION`,
`FILECMD_ALLOCBLOCKS`, `FILECMD_SETEOF`, `FILECMD_ADDENTRY`,
`FILECMD_DELETEENTRY`, `FILECMD_SETTYPE`, `FILECMD_FSSTAT`, `FILECMD_ADDDIR`,
`FILECMD_DELETEDIR`, `OpenFile()`, `CloseFile()`

FILECMD_READENTRY

Read a directory entry by name

Description

This command allows the caller to scan through a list of directory entries looking for a specific entry. The entry is identified by its name. A directory entry is either a file or a subdirectory. Target directory must have been previously opened through `OpenFile()`.

IOInfo`ioi_Command`Set to `FILECMD_READENTRY``ioi_Send.iob_Buffer`

Pointer to a null-terminated filename.

`ioi_Send.iob_Len`Set to `strlen(filename) + 1`.`ioi_Recv.iob_Buffer`Pointer to the `DirectoryEntry` structure.`ioi_Recv.iob_Len`Set to `sizeof(DirectoryEntry)`.**Implementation**

Command defined in Portfolio V27.

Associated Files

<:file:filesystem.h>, <:file:directory.h>, <:kernel:devicecmd.h>

See Also

`FILECMD_GETPATH`, `FILECMD_READDIR`, `FILECMD_SETVERSION`,
`FILECMD_ALLOCBLOCKS`, `FILECMD_SETEOF`, `FILECMD_ADDENTRY`,
`FILECMD_DELETEENTRY`, `FILECMD_SETTYPE`, `FILECMD_FSSTAT`, `FILECMD_ADDDIR`,
`FILECMD_DELETEDIR`, `OpenFile()`, `CloseFile()`

FILECMD_SETEOF

Set the End Of File (EOF)

Description

This command sets the logical end of file for the specified file. EOF can not extend beyond the space allocated to the file via FILECMD_ALLOCBLOCKS, Portfolio Operating System does not support holes in files at this time. EOF is automatically set when the file is written passed its previous EOF. When a file is created EOF is set to zero. It remains zero even though storage is allocated to it. The only time EOF is changed is either through this command or when an actual write occurs extending the logical size of the file. Target file must have been previously opened through `OpenFile()`.

IOInfo`ioi_Command`

Set to FILECMD_SETEOF

`ioi_Offset`

New logical file size in bytes.

Implementation

Command defined in Portfolio V27.

Associated Files

<:file:filesystem.h>, <:kernel:devicecmd.h>

See Also

FILECMD_READENTRY, FILECMD_READDIR, FILECMD_ALLOCBLOCKS,
FILECMD_GETPATH, FILECMD_SETVERSION, FILECMD_ADDENTRY,
FILECMD_DELETEENTRY, FILECMD_SETTYPE, FILECMD_FSSTAT, FILECMD_ADDDIR,
FILECMD_DELETEDIR, `OpenFile()`, `CloseFile()`, `CreateFile()`

FILECMD_SETTYPE

Set file type

Description

This command sets the file type to specified type. Type can not be set for directories (*dir) or system meta files. Only user file types can be manipulated. At file creation file type is set to NULL. Target file must have been previously opened through `OpenFile()`.

IOInfo`ioi_Command`Set to `FILECMD_SETTYPE``ioi_Offset`

New file type.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:file:filesystem.h>`, `<:kernel:devicecmd.h>`

See Also

`FILECMD_READENTRY`, `FILECMD_READDIR`, `FILECMD_ALLOCBLOCKS`,
`FILECMD_GETPATH`, `FILECMD_SETVERSION`, `FILECMD_ADDENTRY`,
`FILECMD_DELETEENTRY`, `FILECMD_SETEOF`, `FILECMD_FSSTAT`, `FILECMD_ADDDIR`,
`FILECMD_DELETEDIR`, `OpenFile()`, `CloseFile()`, `CreateFile()`

FILECMD_SETVERSION

Set revision and version of a file

Description

This command sets a revision and version number for specified file. Target file must have been previously opened through `OpenFile()`.

IOInfo`ioi_Command`Set to `FILECMD_SETVERSION``ioi_Offset`

Version and revision to set. Version is the first byte followed by revision number.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:file:filesystem.h>`, `<:kernel:devicecmd.h>`

See Also

`FILECMD_READENTRY`, `FILECMD_READDIR`, `FILECMD_ALLOCBLOCKS`,
`FILECMD_GETPATH`, `FILECMD_SETEOF`, `FILECMD_ADDENTRY`, `FILECMD_DELETEENTRY`,
`FILECMD_SETTYPE`, `FILECMD_FSSTAT`, `FILECMD_ADDDIR`, `FILECMD_DELETEDIR`,
`OpenFile()`, `CloseFile()`, `CreateFile()`

CMD_BLOCKREAD

Reads data from a block-oriented device.

Description

This command causes the requested device to return data. The device must be "block-oriented". This means that the device data is divided into fixed sized "blocks". The size of the blocks on the device is available from the

.ds_DeviceBlockSize field of the DeviceStatus structure. If the .ds_DeviceBlockSize field is zero, the device is not a block-oriented device.

The data on the device is addressible: each block of data has a "block number", and each request to read data specifies the block number of the first block of data to be read (in ioi_Offset). Block numbers start at zero. However, blocks starting at zero are not necessarily accessible; the first accessible block number is given by the .ds_DeviceBlockStart field in the DeviceStatus structure. The number of blocks on the device is given by the .ds_DeviceBlockCount field in the DeviceStatus structure. This block count includes all blocks, beginning with block zero, even if some of the initial blocks are not accessible. Thus, the accessible block numbers are those between .ds_DeviceBlockStart and .ds_DeviceBlockCount-1, inclusive.

IOInfo

ioi_Command

Set to CMD_BLOCKREAD.

ioi_Recv.iob_Buffer

Pointer to a buffer where the data is to be stored.

ioi_Recv.iob_Len

Set to the size of the buffer pointed to by ioi_Recv.iob_Buffer. No more data than this will be stored in the receive buffer. Less data may be returned, if less data is available from the device, or if the size of the buffer is not a multiple of the device block size. The actual amount of data written into the receive buffer is returned in io_Actual.

ioi_Offset

Set to the block number of the first block of data to be read from the device.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

CMD_BLOCKWRITE, CMD_STREAMREAD, CMD_STREAMWRITE, CDROMCMD_READ

CMD_BLOCKWRITE

Writes data to a block-oriented device.

Description

This command sends data to be stored on the device. The device must be "block-oriented" (see CMD_BLOCKREAD)

IOInfo

ioi_Command

Set to CMD_BLOCKWRITE.

ioi_Send.iob_Buffer

Pointer to a buffer containing the data to be written.

ioi_Send.iob_Len

Set to the size of the buffer pointed to by ioi_Recv.iob_Buffer. Less than the full amount of data in the buffer may be written to the device, if the device does not have the capacity to store the full amount of data, or if the buffer is not a multiple of the device block size. The actual amount of data written to the device is returned in io_Actual.

ioi_Offset

Set to the block number of the first block of data to be written to the device.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

CMD_BLOCKREAD, CMD_STREAMWRITE, CMD_STREAMREAD

CMD_GETMAPINFO

Returns information about the memory-mapping capabilities of a device.

Description

This command causes the requested device to return a `MemMappableDeviceInfo` structure. This structure describes the capabilities of the device to map itself into memory space, allowing software to access the device directly via memory references. The `.mmdi_Flags` field describes the basic memory-mapping capabilities:

MM_MAPPABLE

The device is mappable into memory.

MM_READABLE

The device can be read while mapped, via memory reads.

MM_WRITABLE

The device can be written to while mapped, via memory writes.

MM_EXECUTABLE

Code can be executed from the device while mapped.

MM_EXCLUSIVE

The device supports mapping in "exclusive" mode.
See `CMD_MAPRANGE`.

IOInfo

ioi_Command

Set to `CMD_GETMAPINFO`.

ioi_Recv.iob_Buffer

Pointer to a `MemMappableDeviceInfo` structure.

ioi_Recv.iob_Len

Set to `sizeof(MemMappableDeviceInfo)`.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>, <:kernel:driver.h>

See Also

`CMD_MAPRANGE`, `CMD_UNMAPRANGE`

CMD_MAPRANGE

Requests a device to map itself into memory.

Description

This command causes the requested device to map itself into memory address space. After successful completion of this command, the device can be accessed simply by accessing the appropriate memory addresses. A `MapRangeRequest` structure is sent to the device in the `.ioi_Send` buffer. A `MapRangeResponse` structure is returned from the device in the `.ioi_Recv` buffer.

The `.ioi_Offset` field specifies the starting offset within the device of the first byte to be mapped. The `.mrr_BytesToMap` field of the `MemMapRequest` structure specifies the requested number of bytes to be mapped. More than `.mrr_BytesToMap` bytes may be mapped, but the caller should not depend on this. The memory address where the device has been mapped is returned in the `.mrr_MappedArea` field of the `MemMapResponse` structure.

The bits in the `.mrr_Flags` field of the `MemMapRequest` structure specifies how the mapped area will be used. See `CMD_GETMAPINFO` for a description of the meanings of these bits. Only those bits which are set in the `.mmdi_Flags` field of the device's `MemMappableDeviceInfo` structure may be set in the `.mrr_Flags` field of a `MemMapRequest`. If the `MM_EXCLUSIVE` bit is set in the `MemMapRequest`, the `CMD_MAPRANGE` command will succeed only if the device is not currently mapped. Furthermore, after such a mapping succeeds, other `CMD_MAPRANGE` commands will fail until the first mapping is removed with a `CMD_UNMAPRANGE` command.

IOInfo

`ioi_Command`

Set to `CMD_MAPRANGE`

`ioi_Offset`

Offset within the device of the first byte to be mapped.

`ioi_Send.iob_Buffer`

Pointer to a `MapRangeRequest` structure.

`ioi_Send.iob_Len`

Set to `sizeof(MapRangeRequest)`.

`ioi_Recv.iob_Buffer`

Pointer to a `MapRangeResponse` structure.

`ioi_Recv.iob_Len`

Set to `sizeof(MapRangeResponse)`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:devicecmd.h>`, `<:kernel:driver.h>`

See Also

`CMD_GETMAPINFO`, `CMD_UNMAPRANGE`

CMD_PREFER_FSTYPE

Requests the preferred filesystem type for a device.

Description

This command causes the requested device to return the type of filesystem with which the device would prefer to be formatted.

IOInfo**ioi_Command**

Set to CMD_PREFER_FSTYPE.

ioi_Recv.iob_BufferPointer to a uint32 which receives the filesystem type. Filesystem types are defined in *<:file:discdata.h>*.**ioi_Recv.iob_Len**

Set to sizeof(uint32).

Implementation

Command defined in Portfolio V27.

Associated Files*<:kernel:devicecmd.h>*, *<:file:discdata.h>***Notes**

This command is meaningful only for devices which have the DS_USAGE_FILESYSTEM bit set in the .ds_DeviceUsageFlags file of their DeviceStatus.

See Also

CMD_STATUS

CMD_STATUS

Requests the DeviceStatus information for a device.

Description

This command causes the requested device to return its DeviceStatus information.

IOInfo

ioi_Command

Set to CMD_STATUS.

ioi_Recv.iob_Buffer

Pointer to a DeviceStatus struct.

ioi_Recv.iob_Len

Set to sizeof(DeviceStatus).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

CMD_STREAMREAD

Reads data from a stream-oriented device.

Description

This command reads data from the associated device. The data is read sequentially, as it comes in. The command normally waits until enough data arrives to satisfy the request completely, although a timeout value can be supplied to cause early termination.

IOInfo

`ioi_Command`

Set to `CMD_STREAMREAD`.

`ioi_CmdOptions`

The minimum number of microseconds before a timeout occurs. If more than this amount of time passes between two bytes of data being received, the `IOReq` is returned to the caller. The number of bytes actually read is available in `io_Actual` as usual. If this `ioi_CmdOptions` is set to 0, it means that no timeout is desired.

`ioi_Recv.iob_Buffer`

Pointer to a buffer where the data is to be stored.

`ioi_Recv.iob_Len`

Set to the number of bytes to read into the buffer. No more data than this will be stored in the receive buffer. Less data may be returned, if less data is available from the device or if a timeout occurs. The actual amount of data put into the receive buffer is returned in `io_Actual`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:devicecmd.h>`

See Also

`CMD_STREAMWRITE`, `CMD_BLOCKREAD`, `CMD_BLOCKWRITE`

CMD_STREAMWRITE

Writes data to a stream-oriented device.

Description

This command writes data to the device in a stream-oriented manner.

IOInfo

`ioi_Command`

Set to `CMD_STREAMWRITE`.

`ioi_Send.iob_Buffer`

Pointer to a buffer containing the data to be written.

`ioi_Send.iob_Len`

Set to the number of bytes to output from the send buffer. As much data as possible is written out, although a device can fill up or refuse too much data. The actual number of bytes written is returned in `io_Actual`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:devicecmd.h>`

See Also

`CMD_STREAMREAD`, `CMD_BLOCKREAD`, `CMD_BLOCKWRITE`

CMD_UNMAPRANGE

Requests a device to unmap itself from memory.

Description

This command undoes the effect of a previous `CMD_MAPRANGE` command. After successful completion of this command, a mapping previously set up by a `CMD_MAPRANGE` command is no longer valid, and the memory addresses which previously referred to the device may no longer be accessed. A `MapRangeRequest` structure is sent to the device in the `.ioi_Send` buffer.

The `.ioi_Offset` field must be identical to the `.ioi_Offset` of the `CMD_MAPRANGE` which originally set up the mapping. The `.mrr_BytesToMap` field and the `.mrr_Flags` field of the `MemMapRequest` structure must likewise be identical to the corresponding fields of the of the `MemMapRequest` structure which originally set up the mapping.

IOInfo

`ioi_Command`

Set to `CMD_UNMAPRANGE`.

`ioi_Offset`

Offset within the device of the first byte of the mapping to be undone.

`ioi_Send.iob_Buffer`

Pointer to a `MapRangeRequest` structure.

`ioi_Send.iob_Len`

Set to `sizeof(MapRangeRequest)`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:devicecmd.h>`, `<:kernel:driver.h>`

See Also

`CMD_MAPRANGE`, `CMD_GETMAPINFO`

GFXCMD_EXECUTETECMDS

Submits a list of rendering commands for execution by the Triangle Engine.

Description

This command submits a list of rendering commands to the triangle engine hardware for execution. The commands reside in a buffer managed and filled by the application.

The command accepts a `teRenderInfo` structure, which describes the Bitmaps to be affected (framebuffer and Z-buffer), whether to perform hardware context management, a version header, and a pointer to first triangle engine command to be executed.

teRenderInfo

The `teRenderInfo` structure is used by the triangle engine device driver to determine how to perform the rendering. The structure contains the following fields:

ri_Context

Item number of the `TEContext` Item through which hardware context management should be performed. If set to `GFX_NO_CONTEXT`, context management is not performed.

ri_FrameBuffer

Item number of the Bitmap into which the rendering output of the triangle engine is to be deposited. If set to `GFX_NO_FRAME_BUFFER`, the rendering output of the triangle engine is suppressed.

ri_ZBuffer

Item number of the Bitmap into which the Z-buffering output of the triangle engine is to be deposited. If set to `GFX_NO_Z_BUFFER`, no Z-buffering is performed (note this implies that you must depth-sort your polygons for the imagery to appear correct).

ri_Header

A `CmdHeader` structure, which contains a version number describing which "flavor" of triangle engine command list is being submitted for execution. This is intended to be used by future systems to indicate that an "old" command list requires translation to operate on future hardware. `ri_Header.ch_Version` should be set to `GFX_CMD_LIST_VERSION`.

ri_CmdList

Pointer to the first instruction of the command list.

IOInfo**ioi_Command**

Set to `GFXCMD_EXECUTETECMDS`.

ioi_Send.iob_Buffer

Pointer to a `teRenderInfo` structure.

ioi_Send.iob_Len

Set to `sizeof(struct teRenderInfo)`.

Implementation

Command defined in Portfolio V?? (FIXME: since earliest M2)

Associated Files

`<:kernel:devicecmd.h>, <:graphics:graphics.h>`

See Also

TEContext

HOST_CMD_RECV

Waits for a packet of information from a remote host development system.

Description

This command waits for a 26 byte packet of information to arrive from the remote debugging host.

To receive information from the host, you must specify a particular unit number. Unit numbers 0..127 are reserved for system use. You can use units 128..255 for your own uses. By writing code on the host system, you can send packets of information on any one of these units, and have code running on the 3DO system receive them. This can be very useful during development.

The reserved units are used for things like host file system interfacing, host command-line interface, and so forth. There are higher-level APIs to access these units.

If there are no pending receive requests when a packet comes in from the host, the packet gets buffered by the driver. A fixed number of packets are buffered before overflows start occurring.

IOInfo

ioi_Command

Set to HOST_CMD_RECV

ioi_CmdOptions

Set to the unit number to use. Valid values are in the range 128..255.

ioi_Recv.iob_Buffer

Points to a buffer where the packet of information can be put.

ioi_Recv.iob_Len

Number of bytes available for the packet. This can be in the range 1..26.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOST_CMD_SEND

HOST_CMD_SEND

Sends a packet of information to a remote host development system.

Description

This command sends a 26 byte packet of information to the remote debugging host.

To send information over to the host, you must specify a particular unit number. Unit numbers 0..127 are reserved for system use. You can use units 128..255 for your own uses. By writing code on the host system, you can read packets of information from any one of these units. This can be very useful during development.

The reserved units are used for things like host file system interfacing, host command-line interface, and so forth. There are higher-level APIs to access these units.

IOInfo

ioi_Command

Set to HOST_CMD_SEND

ioi_CmdOptions

Set to the unit number to use. Valid values are in the range 128..255.

ioi_Send.iob_Buffer

Points to a buffer containing the data to send to the host.

ioi_Send.iob_Len

Number of bytes to send to the host. This can be in the range 1..26.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOST_CMD_RECV

HOSTCONSOLE_CMD_GETCMDLINE

Gets command-line input from the remote host.

Description

This command lets you request that the host prompt the user for an input string. When the input string is entered, it is returned to you.

IOInfo

ioi_Command

Set to HOSTCONSOLE_CMD_GETCMDLINE

ioi_Send.iob_Buffer

Pointer to a NULL-terminated string that will be used to prompt the user for input.

ioi_Send.iob_Len

Length of the prompt string, including the NULL-terminator.

ioi_Recv.iob_Buffer

Buffer where the NULL-terminated returned command-line will be put.

ioi_Recv.iob_Len

Maximum length of the returned command-line, including the NULL-terminator.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

HOSTFS_CMD_ALLOCBLOCKS Controls the number of blocks allocated to a file on a remote host file system.

Description

This command lets you add or remove blocks from a file. The blocks are always added to or removed from the end of the file. The number of blocks in a file determines how much data is contained in the file.

IOInfo

ioi_Command

Set to HOSTFS_CMD_ALLOCBLOCKS

ioi_Offset

Set to the number of blocks to add or remove from the file. A positive value specifies blocks to add, while a negative value specifies blocks to remove.

ioi_CmdOptions

Reference token identifying the file to operate on, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_BLOCKREAD

Reads data from an opened file on a remote host file system.

Description

This command lets you read blocks of data from an opened file on a remote file system.

IOInfo

ioi_Command

Set to HOSTFS_CMD_BLOCKREAD

ioi_Offset

Specifies the block number where to start reading. This value must not be greater than the number of blocks currently in the file.

ioi_Recv.iob_Buffer

Buffer where the data should be placed.

ioi_Recv.iob_Len

Number of bytes of data to read. This must be a multiple of the file's block size. The block size can be determined by using the HOSTFS_CMD_STATUS command.

ioi_CmdOptions

Reference token identifying the file to read from, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY, HOSTFS_CMD_CLOSEENTRY, HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR, HOSTFS_CMD_DELETEENTRY, HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR, HOSTFS_CMD_ALLOCBLOCKS, HOSTFS_CMD_MOUNTFS,
HOSTFS_CMD_STATUS, HOSTFS_CMD_FSSTAT, HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF, HOSTFS_CMD_SETTYPE, HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS, HOSTFS_CMD_SETBLOCKSIZE,

HOSTFS_CMD_BLOCKWRITE Writes data to a remote host file system.

Description

This command lets you write blocks of data to a remote host file system.

IOInfo

`ioi_Command`

Set to `HOSTFS_CMD_BLOCKWRITE`

`ioi_Offset`

Specifies the block number where to start writing. This value must not be greater than the number of blocks currently in the file.

`ioi_Send.iob_Buffer`

Buffer where the data to write can be found.

`ioi_Recv.iob_Len`

Number of bytes of data to write. This must be a multiple of the file's block size. The block size can be determined by using the `HOSTFS_CMD_STATUS` command.

`ioi_CmdOptions`

Reference token identifying the file to write to, as obtained using `HOSTFS_CMD_OPENENTRY`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:devicecmd.h>`

See Also

<code>HOSTFS_CMD_OPENENTRY,</code>	<code>HOSTFS_CMD_CLOSEENTRY,</code>	<code>HOSTFS_CMD_CREATEFILE,</code>
<code>HOSTFS_CMD_CREATEDIR,</code>	<code>HOSTFS_CMD_DELETEENTRY,</code>	<code>HOSTFS_CMD_READENTRY</code>
<code>HOSTFS_CMD_READDIR,</code>	<code>HOSTFS_CMD_ALLOCBLOCKS,</code>	<code>HOSTFS_CMD_BLOCKREAD,</code>
<code>HOSTFS_CMD_STATUS,</code>	<code>HOSTFS_CMD_FSSTAT,</code>	<code>HOSTFS_CMD_MOUNTFS,</code>
<code>HOSTFS_CMD_SETEOF,</code>	<code>HOSTFS_CMD_SETTYPE,</code>	<code>HOSTFS_CMD_SETVERSION,</code>
<code>HOSTFS_CMD_DISMOUNTFS,</code>	<code>HOSTFS_CMD_SETBLOCKSIZE,</code>	

HOSTFS_CMD_CLOSEENTRY Concludes use of a reference token.**Description**

This command is used to indicate that a given reference token is no longer needed. The host is then free to reclaim any memory associated with the token. Future attempts to use the token will fail.

IOInfo

ioi_Command

Set to HOSTFS_CMD_CLOSEENTRY

ioi_CmdOptions

Reference token identifying the object to operate on, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_CREATEDIR

Creates a new directory within an existing directory.

Description

This command is used to create a new directory within an existing directory on a remote host file system.

IOInfo

`ioi_Command`

Set to `HOSTFS_CMD_CREATEDIR`

`ioi_Send.iob_Buffer`

Points to the NULL-terminated name of the directory to create,

`ioi_Send.iob_Len`

Number of bytes in the directory name, including the NULL terminator.

`ioi_CmdOptions`

Reference token identifying the directory or file system into which the directory should be created, as obtained using `HOSTFS_CMD_MOUNTFS` or `HOSTFS_CMD_OPENENTRY`.

Return Value

When the directory is successfully created, the `ioi_CmdOptions` field of the `IOReq` item is set to a reference token which uniquely identifies the new directory to the host. You use this token to reference this directory using other commands.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:devicecmd.h>`

See Also

`HOSTFS_CMD_OPENENTRY`, `HOSTFS_CMD_CLOSEENTRY`, `HOSTFS_CMD_CREATEFILE`,
`HOSTFS_CMD_MOUNTFS`, `HOSTFS_CMD_DELETEENTRY`, `HOSTFS_CMD_READENTRY`,
`HOSTFS_CMD_READDIR`, `HOSTFS_CMD_ALLOCBLOCKS`, `HOSTFS_CMD_BLOCKREAD`,
`HOSTFS_CMD_STATUS`, `HOSTFS_CMD_FSSTAT`, `HOSTFS_CMD_BLOCKWRITE`,
`HOSTFS_CMD_SETEOF`, `HOSTFS_CMD_SETTYPE`, `HOSTFS_CMD_SETVERSION`,
`HOSTFS_CMD_DISMOUNTFS`, `HOSTFS_CMD_SETBLOCKSIZE`,

HOSTFS_CMD_CREATEFILE

Creates a new file within an existing directory.

Description

This command is used to create a new file within an existing directory on a remote host file system.

IOInfo**ioi_Command**

Set to HOSTFS_CMD_CREATEFILE

ioi_Send.iob_Buffer

Points to the NULL-terminated name of the file to create.

ioi_Send.iob_Len

Number of bytes in the file name, including the NULL terminator.

ioi_CmdOptions

Reference token identifying the directory or file system into which the file should be created, as obtained using HOSTFS_CMD_MOUNTFS or HOSTFS_CMD_OPENENTRY.

Return Value

When the file is successfully created, the ioi_CmdOptions field of the IOReq item is set to a reference token which uniquely identifies the new file to the host. You use this token to reference this file using other commands.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY, HOSTFS_CMD_CLOSEENTRY, HOSTFS_CMD_MOUNTFS,
HOSTFS_CMD_CREATEDIR, HOSTFS_CMD_DELETEENTRY, HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR, HOSTFS_CMD_ALLOCBLOCKS, HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS, HOSTFS_CMD_FSSTAT, HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF, HOSTFS_CMD_SETTYPE, HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS, HOSTFS_CMD_SETBLOCKSIZE,

HOSTFS_CMD_DELETEENTRY Deletes an entry within a remote file system.**Description**

This command is used to delete an entry from a remote host file system. The command works for both files and directories. Directories can only be deleted if they are empty.

IOInfo

ioi_Command

Set to HOSTFS_CMD_DELETEENTRY

ioi_Send.iob_Buffer

Points to the NULL-terminated name of the object to delete,

ioi_Send.iob_Len

Number of bytes in the object name, including the NULL terminator.

ioi_CmdOptions

Reference token identifying the directory containing the object to delete, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_DISMOUNTFS

Requests that a file system be dismounted.

Description

This command causes a remote host file system to be dismounted and become unavailable for use.

IOInfo

ioi_Command

Set to HOSTFS_CMD_DISMOUNTFS

ioi_CmdOptions

Set to the reference token obtained when the file system was first mounted using the HOSTFS_CMD_MOUNTFS command.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY, HOSTFS_CMD_CLOSEENTRY, HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR, HOSTFS_CMD_DELETEENTRY, HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR, HOSTFS_CMD_ALLOCBLOCKS, HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS, HOSTFS_CMD_FSSTAT, HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF, HOSTFS_CMD_SETTYPE, HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_MOUNTFS, HOSTFS_CMD_SETBLOCKSIZE,

HOSTFS_CMD_FSSTAT

Obtains information about a remote host file system.

Description

This command returns information about a remote host file system. The information is returned in a `FileSystemStat` structure, as defined in `<:file:filesystem.h>`

IOInfo

`ioi_Command`

Set to `HOSTFS_CMD_FSSTAT`

`ioi_Recv.iob_Buffer`

Points to a `FileSystemStat` structure where the information about the remote file system will be stored.

`ioi_Recv.iob_Len`

Set to `sizeof(FileSystemStat)`.

`ioi_CmdOptions`

Reference token for any object on the remote file system to get information on.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:file:filesystem.h>`, `<:kernel:devicecmd.h>`

See Also

<code>HOSTFS_CMD_OPENENTRY,</code>	<code>HOSTFS_CMD_CLOSEENTRY,</code>	<code>HOSTFS_CMD_CREATEFILE,</code>
<code>HOSTFS_CMD_CREATEDIR,</code>	<code>HOSTFS_CMD_DELETEENTRY,</code>	<code>HOSTFS_CMD_READENTRY</code>
<code>HOSTFS_CMD_READDIR,</code>	<code>HOSTFS_CMD_ALLOCBLOCKS,</code>	<code>HOSTFS_CMD_BLOCKREAD,</code>
<code>HOSTFS_CMD_STATUS,</code>	<code>HOSTFS_CMD_MOUNTFS,</code>	<code>HOSTFS_CMD_BLOCKWRITE,</code>
<code>HOSTFS_CMD_SETEOF,</code>	<code>HOSTFS_CMD_SETTYPE,</code>	<code>HOSTFS_CMD_SETVERSION,</code>
<code>HOSTFS_CMD_DISMOUNTFS,</code>	<code>HOSTFS_CMD_SETBLOCKSIZE,</code>	

HOSTFS_CMD_MOUNTFS

Requests that a file system be mounted on a remote host.

Description

This command requests that the remote debugging host mount a file system and return a reference token uniquely identifying it.

IOInfo

`ioi_Command`

Set to `HOSTFS_CMD_MOUNTFS`

`ioi_Offset`

File system number to mount. The host keeps mountable file systems in a numbered list starting at 0. You can mount any of these file systems. Asking to mount a non-existing file system will result in an error code being returned.

`ioi_Recv.iob_Buffer`

Points to a buffer where the Host will put the name of the file system that was mounted. This pointer may be NULL in which case no name is returned.

`ioi_Recv.iob_Len`

Number of bytes available for the file system name, including the NULL terminator.

Return Value

When the file system is successfully mounted, the `ioi_CmdOptions` field of the `IOReq` item is set to a reference token which uniquely identifies the file system to the host. You use this token to reference objects within this file system.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

<code>HOSTFS_CMD_OPENENTRY,</code>	<code>HOSTFS_CMD_CLOSEENTRY,</code>	<code>HOSTFS_CMD_CREATEFILE,</code>
<code>HOSTFS_CMD_CREATEDIR,</code>	<code>HOSTFS_CMD_DELETEENTRY,</code>	<code>HOSTFS_CMD_READENTRY</code>
<code>HOSTFS_CMD_READDIR,</code>	<code>HOSTFS_CMD_ALLOCBLOCKS,</code>	<code>HOSTFS_CMD_BLOCKREAD,</code>
<code>HOSTFS_CMD_STATUS,</code>	<code>HOSTFS_CMD_FSSTAT,</code>	<code>HOSTFS_CMD_BLOCKWRITE,</code>
<code>HOSTFS_CMD_SETEOF,</code>	<code>HOSTFS_CMD_SETTYPE,</code>	<code>HOSTFS_CMD_SETVERSION,</code>
<code>HOSTFS_CMD_DISMOUNTFS,</code>	<code>HOSTFS_CMD_SETBLOCKSIZE,</code>	

HOSTFS_CMD_OPENENTRY

Obtains a reference token for an object within a remote file system.

Description

This command lets you request a reference token for a named object within a remote file system. The named object is relative to the object to which the reference token points to. That is, if the reference token was obtained from HOSTFS_CMD_MOUNTFS, then the named object is in the root of the remote file system. If the reference token was obtained from a previous call to HOSTFS_CMD_OPENENTRY, then the name is relative to a directory within the remote file system.

IOInfo

ioi_Command

Set to HOSTFS_CMD_OPENENTRY

ioi_Send.iob_Buffer

Pointer to the NULL-terminated name of the object to obtain the reference token for. This name is relative to the supplied reference token.

ioi_Send.iob_Len

Number of bytes in the object name, including the NULL terminator.

ioi_CmdOptions

Reference token for the directory or file system that contains the object to open.

Return Value

When the object is successfully opened, the ioi_CmdOptions field of the IOReq item is set to a reference token which uniquely identifies the desired object to the host. You use this token to reference this object using other commands.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_READDIR

Obtains information about an object within a remote host file system.

Description

This command returns information about a particular object within a remote host file system. The information is returned in a DirectoryEntry structure as defined in *<:file:directory.h>*.

This command is very similar to HOSTFS_CMD_READENTRY, except that the object to get information on is specified by index number within a directory, instead of by name of object within the directory. This is the command to use when listing all files in a directory.

ioi_Command

Set to HOSTFS_CMD_READDIR

ioi_Offset

The index of the object to get information on. Index values start at 1, not 0.

ioi_Recv.iob_Buffer

Points to a DirectoryEntry structure where the information about the object will be stored.

ioi_Recv.iob_Len

Set to sizeof(DirectoryEntry).

ioi_CmdOptions

Reference token identifying the directory containing the object to get information on, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:file:directory.h>, *<:kernel:devicecmd.h>*

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_READENTRY

Obtains information about an object within a remote host file system.

Description

This command returns information about a particular object within a remote host file system. The information is returned in a DirectoryEntry structure as defined in *<:file:directory.h>*

IOInfo

ioi_Command

Set to HOSTFS_CMD_READENTRY

ioi_Send.iob_Buffer

Points to the NULL-terminated name of the object to get information on.

ioi_Send.iob_Len

Number of bytes in the object name, including the NULL terminator.

ioi_Recv.iob_Buffer

Points to a DirectoryEntry structure where the information about the object will be stored.

ioi_Recv.iob_Len

Set to sizeof(DirectoryEntry).

ioi_CmdOptions

Reference token identifying the directory containing the object to get information on, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:file:directory.h>, *<:kernel:devicecmd.h>*

See Also

HOSTFS_CMD_OPENENTRY, HOSTFS_CMD_CLOSEENTRY, HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR, HOSTFS_CMD_DELETEENTRY, HOSTFS_CMD_MOUNTFS
HOSTFS_CMD_READDIR, HOSTFS_CMD_ALLOCBLOCKS, HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS, HOSTFS_CMD_FSSTAT, HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF, HOSTFS_CMD_SETTYPE, HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS, HOSTFS_CMD_SETBLOCKSIZE,

HOSTFS_CMD_RENAMEENTRY Renames an object on a remote host file system.**Description**

This command lets you rename a file or directory located on a remote file system. You supply the new name of the object.

IOInfo

ioi_Command

Set to HOSTFS_CMD_RENAMEENTRY

ioi_Send.iob_Buffer

Points to the NULL-terminated new name of the object to delete,

ioi_Send.iob_Len

Number of bytes in the new object name, including the NULL terminator.

ioi_CmdOptions

Reference token identifying the object the object being renamed, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V30.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_SETBLOCKSIZE Sets the device block size for a particular IOReq dealing with a file on a remote host file system.

Description

This command lets you set the block size used when reading or writing data using the current IOReq. If this command is not set, then the default size is 8K. The proper value to be set here is determined by sending a HOSTFS_CMD_STATUS command to the opened entry, and extracting the block size from the FileStatus structure.

IOInfo

ioi_Command

Set to HOSTFS_CMD_SETBLOCKSIZE.

ioi_Offset

The block size to use for this IOReq.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_MOUNTFS,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_MOUNTFS,	

HOSTFS_CMD_SETEOF

Sets the logical byte count of a file on a remote host file system.

Description

Every file keeps a count of the number of valid bytes the file contains. This command lets you set this count for an opened file on a remote host file system.

IOInfo

ioi_Command

Set to HOSTFS_CMD_SETEOF

ioi_Offset

Specifies the byte count to set for the opened file.

ioi_CmdOptions

Reference token identifying the file to set the size off, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY, HOSTFS_CMD_CLOSEENTRY, HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR, HOSTFS_CMD_DELETEENTRY, HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR, HOSTFS_CMD_ALLOCBLOCKS, HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS, HOSTFS_CMD_FSSTAT, HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_MOUNTFS, HOSTFS_CMD_SETTYPE, HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS, HOSTFS_CMD_SETBLOCKSIZE,

HOSTFS_CMD_SETTYPE

Sets the four byte file type for an object on a remote host file system.

Description

Every file system object has a four byte file type value. This command lets you set this value for an opened entry on a remote host file system.

IOInfo

ioi_Command

Set to HOSTFS_CMD_SETTYPE

ioi_Offset

Set to the four byte type value.

ioi_CmdOptions

Reference token identifying the object to set the type of, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

HOSTFS_CMD_SETVERSION

Set the version and revision codes for an opened entry on a remote host file system.

Description

Every file system object has a version and revision code associated with it. This command lets you set these values for an opened entry on a remote host file system.

IOInfo

ioi_Command

Set to HOSTFS_CMD_SETVERSION

ioi_Offset

Specifies the version and revision code for the object. The two values are 8 bit wide and pack like: (version << 8) | revision.

ioi_CmdOptions

Reference token identifying the object to set the version of, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>

See Also

HOSTFS_CMD_OPENENTRY, HOSTFS_CMD_CLOSEENTRY, HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR, HOSTFS_CMD_DELETEENTRY, HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR, HOSTFS_CMD_ALLOCBLOCKS, HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_STATUS, HOSTFS_CMD_FSSTAT, HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF, HOSTFS_CMD_SETTYPE, HOSTFS_CMD_MOUNTFS,
HOSTFS_CMD_DISMOUNTFS, HOSTFS_CMD_SETBLOCKSIZE,

HOSTFS_CMD_STATUS

Obtains information about an opened entry on a remote host file system.

Description

This command returns information about an opened entry within a remote host file system. The information is returned in a FileStatus structure as defined in *<:file:filesystem.h>*

IOInfo

ioi_Command

Set to HOSTFS_CMD_STATUS

ioi_Recv.iob_Buffer

Points to a FileStatus structure where the information on the entry will be stored.

ioi_Recv.iob_Len

Set to sizeof(FileStatus)

ioi_CmdOptions

Reference token identifying the object to get information on, as obtained using HOSTFS_CMD_OPENENTRY.

Implementation

Command defined in Portfolio V27.

Associated Files

<:file:filesystem.h>, *<:kernel:devicecmd.h>*

See Also

HOSTFS_CMD_OPENENTRY,	HOSTFS_CMD_CLOSEENTRY,	HOSTFS_CMD_CREATEFILE,
HOSTFS_CMD_CREATEDIR,	HOSTFS_CMD_DELETEENTRY,	HOSTFS_CMD_READENTRY
HOSTFS_CMD_READDIR,	HOSTFS_CMD_ALLOCBLOCKS,	HOSTFS_CMD_BLOCKREAD,
HOSTFS_CMD_MOUNTFS,	HOSTFS_CMD_FSSTAT,	HOSTFS_CMD_BLOCKWRITE,
HOSTFS_CMD_SETEOF,	HOSTFS_CMD_SETTYPE,	HOSTFS_CMD_SETVERSION,
HOSTFS_CMD_DISMOUNTFS,	HOSTFS_CMD_SETBLOCKSIZE,	

MPEGVIDEOCMD_CONTROL

Permits application control over MPEG video decoding.

Description

This command submits one or more control request to the MPEG video device. A TagArg list is used to set various parameters and modes.

IOInfo

ioi_Command

Set to MPEGVIDEOCMD_CONTROL

ioi_Send.iob_Buffer

Pointer to a CODECDeviceStatus structure.

ioi_Send.iob_Len

Set to sizeof(CODECDeviceStatus);

The following control requests may be submitted in the codec_TagArg list in the CODECDeviceStatus structure.

VID_CODEC_TAG_HSIZE

This request sets the horizontal output size. The associated ta_Arg should be set to the horizontal output size and should be a multiple of 16.

VID_CODEC_TAG_VSIZE

This request sets the vertical output size. The associated ta_Arg should be set to the vertical output size and should be a multiple of 16.

VID_CODEC_TAG_DEPTH

This request allows selection of 16 or 32 bit output formats. The associated ta_Arg should be set to 16 or 32.

VID_CODEC_TAG_M2MODE

This request selects M2 output modes and should ALWAYS be set for M2 titles.

VID_CODEC_TAG_KEYFRAMES

This request puts the device into I frame only mode. After this request is sent the device will only decode and return MPEG I pictures. Other pictures in the input bitstream are skipped. Use VID_CODEC_TAG_PLAY to return to normal play mode.

VID_CODEC_TAG_PLAY

This request puts the device into normal play mode. After this request is sent the device will decode and return all MPEG pictures in the input bitstream. Used to exit I frame only mode.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>, <:device:mpegvideo.h>

See Also

MPEGVIDEOCMD_READ, MPEGVIDEOCMD_WRITE

MPEGVIDEOCMD_READ

Submits a bitmap item for the MPEG video device to place a decoded picture.

Description

This command submits a bitmap item to the MPEG video device. The device then places a decoded frame of MPEG video into the bitmap's buffer.

IOInfo

ioi_Command

Set to MPEGVIDEOCMD_READ.

ioi_Send.iob_Buffer

Pointer to a bitmap Item. The bitmap item must have been created using BMTAG_MPEGABLE set to TRUE.

ioi_Send.iob_Len

Set to sizeof(Item);

When the request has completed the following IOReq fields may be set:

io_Flags

The 0x80000000 bit is set if a presentation time stamp is valid for this read request.

io_Extension[0]

Contains the presentation time stamp for this read request if the above bit is set. Can be accessed using the FMVGetPTSValue macro.

io_Info.ioi_CmdOptions

Contains the user data associated with this read request. Can be accessed using the FMVGetUserData macro.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>, <:device:mpegvideo.h>, <:graphics:bitmap.h>

See Also

MPEGVIDEOCMD_WRITE, MPEGVIDEOCMD_CONTROL

MPEGVIDEOCMD_WRITE

Submits a buffer of MPEG video data for decoding by the MPEG video device.

Description

This command submits a buffer containing MPEG video data to the MPEG video device for decoding. Presentation time stamp (PTS) information and user data may be optionally sent with the request.

IOInfo

ioi_Command

Set to MPEGVIDEOCMD_WRITE.

ioi_CmdOptions

Optional pointer to an FMVIOReqOptions structure containing presentation time stamp (PTS) and user information. This data flows through the decoder and is output with the corresponding read request.

ioi_Send.iob_Buffer

Pointer to a buffer of MPEG video data.

ioi_Send.iob_Len

Set to the size of the MPEG video data buffer.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:devicecmd.h>, <:device:mpegvideo.h>

See Also

MPEGVIDEOCMD_READ, MPEGVIDEOCMD_CONTROL

SER_CMD_BREAK

Sends a break signal over the serial line.

Description

This command sends a break signal (serial line held low for an extended period). The length of the break signal can be specified by the client.

IOInfo

ioi_Command

Set to SER_CMD_BREAK.

ioi_CmdOptions

Specifies the duration of the break signal in microseconds. If this value is set to 0, a default is assumed.

Implementation

Command defined in Portfolio V29.

Associated Files

<:kernel:devicecmd.h>, *<:device:serial.h>*

See Also

CMD_STREAMWRITE, CMD_STREAMREAD

SER_CMD_GETCONFIG

Determines the current serial port settings.

Description

This command returns the current serial port settings in a SerConfig structure.

IOInfo

ioi_Command

Set to SER_CMD_GETCONFIG.

ioi_Recv.iob_Buffer

Points to a SerConfig structure as defined in *<:device:serial.h>*, where the current serial configuration parameters will be stored.

ioi_Recv.iob_Len

Set to sizeof(SerConfig).

Implementation

Command defined in Portfolio V29.

Associated Files

<:kernel:devicecmd.h>, *<:device:serial.h>*

See Also

SER_CMD_SETCONFIG

SER_CMD_SETCONFIG

Sets the configuration of the serial port.

Description

This command lets you set various serial attributes. Sending this command has the effect of resetting the serial port.

The default configuration for a serial port is:

```
sc_BaudRate      = 57600;  
sc_Handshake     = SER_HANDSHAKE_NONE;  
sc_WordLength   = SER_WORDLENGTH_8;  
sc_Parity       = SER_PARITY_NONE;  
sc_StopBits     = SER_STOPBITS_1;  
sc_OverflowBufferSize = 0;
```

The `sc_OverflowBufferSize` field lets you specify the size in bytes of a buffer maintained by the driver to deal with read overflow conditions. This happens when data is coming into the serial port and there is no `IOReq` present to read the data. It is more efficient to use `IOReq` double-buffering rather than to rely on the driver to buffer data.

IOInfo

`ioi_Command`
Set to `SER_CMD_SETCONFIG`.

`ioi_Send.iob_Buffer`
Points to an initialized `SerConfig` structure as defined in `<:device:serial.h>`, specifying the new serial configuration parameters.

`ioi_Send.iob_Len`
Set to `sizeof(SerConfig)`.

Implementation

Command defined in Portfolio V29.

Associated Files

`<:kernel:devicecmd.h>`, `<:device:serial.h>`

See Also

`SER_CMD_GETCONFIG`

SER_CMD_SETDTR

Controls the state of the serial DTR line.

Description

This command lets you control the state of the serial DTR line. The line can be made to go high or low. When the device is first initialized, the DTR line is automatically held high. When the device is shutting down, DTR is automatically dropped low.

IOInfo**ioi_Command**

Set to SER_CMD_SETDTR.

ioi_CmdOptions

Set this field to 1 to set DTR or 0 to clear DTR.

Implementation

Command defined in Portfolio V29.

Associated Files

<:kernel:devicecmd.h>, <:device:serial.h>

See Also

SER_CMD_STATUS, SER_CMD_SETRTS

SER_CMD_SETLOOPBACK

Controls the state of automatic serial loopback mode.

Description

This command lets you control serial loopback, which pumps outgoing data back into the serial port. This is sometimes useful for diagnostic and for developing code without having a terminal hooked up.

IOInfo

ioi_Command

Set to SER_CMD_SETLOOPBACK.

ioi_CmdOptions

Set to 1 to turn loopback mode on or 0 to set loopback mode off.

Implementation

Command defined in Portfolio V29.

Associated Files

<:kernel:devicecmd.h>, *<:device:serial.h>*

See Also

SER_CMD_STATUS, SER_CMD_SETRTS, SER_CMD_SETDTR

SER_CMD_SETRTS

Controls the state of the serial RTS line.

Description

This command lets you control the state of the serial RTS line. The line can be made to go high or low.

This command can only be used when the driver is configured for no handshaking or software handshaking. The driver takes over the RTS line when hardware handshaking is enabled.

IOInfo**ioi_Command**

Set to SER_CMD_SETRTS.

ioi_CmdOptions

Set this field to 1 to set RTS or 0 to clear RTS.

Implementation

Command defined in Portfolio V29.

Associated Files

<:kernel:devicecmd.h>, <:device:serial.h>

See Also

SER_CMD_STATUS, SER_CMD_SETDTR, SER_CMD_SETLOOPBACK

SER_CMD_STATUS

Gets the state of various serial attributes.

Description

This command returns information about the current state of the serial port. This includes the number of errors encountered since the last reset, as well as the state of certain serial lines.

IOInfo**ioi_Command**

Set to SER_CMD_STATUS.

ioi_Recv.iob_BufferPoints to a SerStatus structure as defined in *<:device:serial.h>*, where the status information will be stored.**ioi_Recv.iob_Len**

Set to sizeof(SerStatus).

Implementation

Command defined in Portfolio V29.

Associated Files*<:kernel:devicecmd.h>*, *<:device:serial.h>***See Also**

SER_CMD_SETRTS, SER_CMD_SETDTR, SER_CMD_SETLOOPBACK, SER_CMD_WAITEVENT

SER_CMD_WAITEVENT

Waits for serial events to occur.

Description

This command lets you wait for specific events to occur. You can wait for multiple events to occur. The IOReq is returned to you when any of the events occur, and the io_Actual field specifies which events occurred.

IOInfo**ioi_Command**

Set to SER_CMD_WAITEVENT.

ioi_CmdOptionsSpecifies a bitmask of the events to wait for. This mask is constructed using the various SER_EVENT_* constants defined in *<:device:serial.h>*.**Implementation**

Command defined in Portfolio V29.

Associated Files*<:kernel:devicecmd.h>*, *<:device:serial.h>***See Also**

SER_CMD_STATUS

TIMERCMD_DELAYUNTIL_USEC Waits for a given time.

Description

This command does nothing but wait for a given time to arrive. When that time arrives, your IOReq is returned to you. If you ask to wait for a time that has already passed, your IOReq is returned immediately.

IOInfo

ioi_Command

Set to TIMERCMD_DELAYUNTIL_USEC

ioi_Send.iob_Buffer

Pointer to a TimeVal structure specifying the time to wait for.

ioi_Send.iob_Len

Set to sizeof(TimeVal).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_GETTIME_USEC, TIMERCMD_SETTIME_USEC, TIMERCMD_DELAY_USEC,
TIMERCMD_METRONOME_USEC

TIMERCMD_DELAYUNTIL_VBL Waits for the system VBL count to reach a specific number.

Description

This command does nothing but wait for the system's vertical blank counter to reach a specific number. When that number passes, your IOReq is returned to you. This is a simple way to wait for a specific time to arrive. If you ask to wait for a count that has already passed, your IOReq is returned immediately.

IOInfo

ioi_Command

Set to TIMERCMD_DELAYUNTIL_VBL

ioi_Offset

Specifies the vertical blank count value to wait for.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_GETTIME_VBL, TIMERCMD_SETTIME_VBL, TIMERCMD_DELAY_VBL,
TIMERCMD_METRONOME_VBL

TIMERCMD_DELAY_USEC

Waits for a fixed amount of time to pass.

Description

This command does nothing but wait for a specific amount of time to pass. When that amount of time passes, your IOReq is returned to you. This is a simple way to wait for an amount of time to pass.

IOInfo

ioi_Command

Set to TIMERCMD_DELAY_USEC

ioi_Send.iob_Buffer

Pointer to a TimeVal structure specifying the amount of time to wait for.

ioi_Send.iob_Len

Set to sizeof(TimeVal).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_GETTIME_USEC, TIMERCMD_SETTIME_USEC, TIMERCMD_DELAYUNTIL_USEC,
TIMERCMD_METRONOME_USEC

TIMERCMD_DELAY_VBL

Waits for a fixed number of vertical blanking intervals.

Description

This command does nothing but wait for a specific number of vertical blanking intervals to pass. When that number passes, your IOReq is returned to you. This is a simple way to wait for an amount of time to pass.

There are 60 VBLs per second on NTSC systems and 50 per second on PAL systems. Therefore, waiting for 60 VBLs will cause a 1 second delay on NTSC systems, and a 1.2 second delay on PAL systems.

IOInfo**ioi_Command**

Set to TIMERCMD_DELAY_VBL

ioi_Offset

Specifies the number of vertical blanking intervals to wait for.

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See AlsoTIMERCMD_GETTIME_VBL, TIMERCMD_SETTIME_VBL, TIMERCMD_DELAYUNTIL_VBL,
TIMERCMD_METRONOME_VBL

TIMERCMD_GETTIME_USEC Returns the current system time.

Description

This command returns the current system time.

IOInfo

ioi_Command

Set to TIMERCMD_GETTIME_USEC

ioi_Recv.iob_Buffer

Pointer to a TimeVal structure to receive the current time.

ioi_Recv.iob_Len

Set to sizeof(TimeVal).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_SETTIME_USEC, TIMERCMD_DELAY_USEC, TIMERCMD_DELAYUNTIL_USEC,
TIMERCMD_METRONOME_USEC

TIMERCMD_GETTIME_VBL

Returns the current system time in vertical blanking intervals.

Description

This command returns the current system time counted in VBL units. There are 60 VBLs per second on an NTSC system and 50 VBLs per second on a PAL system.

IOInfo

ioi_Command

Set to TIMERCMD_GETTIME_VBL

ioi_Recv.iob_Buffer

Pointer to a TimeValVBL variable where the VBL count will be put.

ioi_Recv.iob_Len

Set to sizeof(TimeValVBL).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_SETTIME_VBL, TIMERCMD_DELAY_VBL, TIMERCMD_DELAYUNTIL_VBL,
TIMERCMD_METRONOME_VBL

TIMERCMD_METRONOME_USEC

Requests to be signalled at regular intervals of time.

Description

This command causes a signal to be sent to your task on a regular basis. This is very useful to deal with operations that must occur consistently at regular time intervals. Once you have submitted an IOReq using this command, you will start receiving signals. In order to stop the signals, you must abort the IOReq using `AbortIO()`.

IOInfo

`ioi_Command`

Set to `TIMERCMD_METRONOME_USEC`

`ioi_Send.iob_Buffer`

Pointer to a `TimeVal` structure that specifies the amount of time between each signal sent to your task.

`ioi_Send.iob_Len`

Set to `sizeof(TimeVal)`.

`ioi_CmdOptions`

The signal mask specifying which signals to send to your task. This is typically the return value of `AllocSignal(0)`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:time.h>`, `<:kernel:devicecmd.h>`

See Also

`TIMERCMD_GETTIME_USEC`, `TIMERCMD_SETTIME_USEC`, `TIMERCMD_DELAYUNTIL_USEC`, `TIMERCMD_DELAY_USEC`, `AllocSignal()`

TIMERCMD_METRONOME_VBL Requests to be signalled at regular intervals of time.

Description

This command causes a signal to be sent to your task on a regular basis. This is very useful to deal with operations that must occur consistently at regular time intervals. Once you have submitted an IOReq using this command, you will start receiving signals. In order to stop the signals, you must abort the IOReq using `AbortIO()`

IOInfo

`ioi_Command`

Set to `TIMERCMD_METRONOME_VBL`

`ioi_Offset`

Specifies the number of vertical blanking intervals between each signal sent to your task.

`ioi_CmdOptions`

The signal mask specifying which signals to send to your task. This is typically the return value of `AllocSignal(0)`.

Implementation

Command defined in Portfolio V27.

Associated Files

`<:kernel:time.h>`, `<:kernel:devicecmd.h>`

See Also

`TIMERCMD_GETTIME_VBL`, `TIMERCMD_SETTIME_VBL`, `TIMERCMD_DELAYUNTIL_VBL`,
`TIMERCMD_DELAY_VBL`, `AllocSignal()`

TIMERCMD_SETTIME_USEC Sets the current system time.

Description

This command lets you set the current system time.

IOInfo

ioi_Command

Set to TIMERCMD_SETTIME_USEC

ioi_Send.iob_Buffer

Pointer to a TimeVal structure specifying the new system time.

ioi_Send.iob_Len

Set to sizeof(TimeVal).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_GETTIME_USEC, TIMERCMD_DELAY_USEC, TIMERCMD_DELAYUNTIL_USEC,
TIMERCMD_METRONOME_USEC

TIMERCMD_SETTIME_VBL

Sets the current system time in vertical blanking intervals.

Description

This command lets you set the current value of the system VBL counter.

IOInfo

ioi_Command

Set to TIMERCMD_SETTIME_VBL

ioi_Send.iob_Buffer

Pointer to a TimeValVBL variable specifying the new VBL count.

ioi_Send.iob_Len

Set to sizeof(TimeValVBL).

Implementation

Command defined in Portfolio V27.

Associated Files

<:kernel:time.h>, <:kernel:devicecmd.h>

See Also

TIMERCMD_GETTIME_VBL, TIMERCMD_DELAY_VBL, TIMERCMD_DELAYUNTIL_VBL,
TIMERCMD_METRONOME_VBL

Chapter 7

Event Broker Calls

This section presents the reference documentation for the Event Broker and associated link libraries.

GetControlPad

Gets control pad events.

Synopsis

```
Err GetControlPad(int32 padNumber, bool wait,  
                  ControlPadEventData *data);
```

Description

This is a convenience call that allows a task to easily monitor events on controller pads. This function specifies a controller pad to monitor, specifies whether the routine should return immediately or wait until something happens on the controller pad, and provides a data structure for data from the controller pad.

When the function executes, it either returns immediately with event information or waits until there is a change in the controller pad before returning. If an event has occurred, the task must check the ControlPadEventData data structure for details about the event.

Arguments

padNumber

Sets the number of the generic controller pad on the control port (i.e., the first, second, third, and so on, pad in the control port daisy chain, counting out from the 3DO unit) that the task wants to monitor. The first pad is 1, the second is 2, and so on.

wait

A boolean value that specifies the event broker's response. If it is TRUE (a nonzero value), the event broker waits along with the task until an event occurs on the specified pad and only returns with data when there is a change in the pad. If it is FALSE (zero), the event broker immediately returns with the status of the pad.

data

A pointer to a ControlPadEventData data structure to receive the returned control pad data.

Return Value

This function returns 1 if an event has occurred on the pad, 0 if no event has occurred on the pad, or a negative error code if a problem occurred while retrieving an event.

Implementation

Link library call implemented in libeventbroker.a V20.

Associated Files

<:misc:event.h>, libeventbroker.a

See Also

InitEventUtility(), GetMouse(), KillEventUtility()

GetMouse

Gets mouse events.

Synopsis

```
Err GetMouse(int32 mouseNumber, bool wait,  
             MouseEventData *data);
```

Description

This convenience call allows a task to easily monitor mouse events.

This function is similar to `GetControlPad()` but gets events from a specified mouse instead of a specified controller pad. It specifies a mouse to monitor, specifies whether the call should return immediately or wait until something happens on the mouse, and provides a data structure for data from the mouse.

When the function executes, it either returns immediately with event information or waits there is a change in the mouse before returning. If an event has occurred, the task must check the `MouseEventData` data structure for details about the event.

Arguments

`mouseNumber`

Sets the number of the generic mouse on the control port (i.e., the first, second, third, and so on, mouse in the control port daisy chain, counting out from the 3DO unit) that the task wants to monitor. The first mouse is 1, the second is 2, and so on.

`wait`

A boolean value that specifies the event broker's response. If it is `TRUE` (a nonzero value), the event broker waits along with the task until an event occurs on the specified pad and only returns with data when there is a change in the pad. If it is `FALSE` (zero), the event broker immediately returns with the status of the pad.

`data`

A pointer to a `MouseEventData` data structure to receive the returned mouse data.

Return Value

This function returns 1 if an event has occurred on the mouse, 0 if no event has occurred on the mouse, or a negative error code if an error occurred when attempting to retrieve an event.

Implementation

Link library call implemented in `libeventbroker.a` V20.

Associated Files

`<:misc:event.h>`, `libeventbroker.a`

See Also

`InitEventUtility()`, `GetControlPad()`, `KillEventUtility()`

InitEventUtility

Connects task to the event broker.

Synopsis

```
Err InitEventUtility(int32 numControlPads, int32 numMice,  
                    int32 focusListener);
```

Description

This convenience call allows a task to easily monitor events on controller pads or mice. This function connects the task to the event broker, sets the task's focus interest, and determines how many controller pads and mice the task wants to monitor.

The function creates a reply port and a message, sends a configuration message to the event broker (which asks the event broker to report appropriate mouse and controller pad events), and deals with the event broker's configuration reply.

Arguments

numControlPads

The number of controller pads to monitor.

numMice

The number of mice to monitor.

focusListener

The focus of the task when it is connected as a listener. If the value is nonzero, the task is connected as a focus-dependent listener. If the value is zero, the task is connected as a focus-independent listener.

Return Value

Returns ≥ 0 if all went well or a negative error code if an error occurred.

Implementation

Link library call implemented in libeventbroker.a V20.

Associated Files

`<:misc:event.h>`, libeventbroker.a

See Also

GetControlPad(), GetMouse(), KillEventUtility()

KillEventUtility

Disconnects a task from the event broker.

Synopsis

```
Err KillEventUtility(void);
```

Description

This function disconnects a task that was connected to the event broker with the `InitEventUtility()` function. When executed, the function deletes the reply port, and frees all resources used for the connection.

Return Value

Returns ≥ 0 if all went well or a negative error code if an error occurred.

Implementation

Link library call implemented in `libeventbroker.a` V20.

Associated Files

`<:misc:event.h>`, `libeventbroker.a`

See Also

`GetControlPad()`, `GetMouse()`, `InitEventUtility()`

Chapter 8

File Folio Calls

This section presents the reference documentation for the File folio and associated link libraries.

ChangeDirectory

Changes the current directory.

Synopsis

```
Item ChangeDirectory(const char *path);
```

Description

Changes the current task's working directory to the absolute or relative location specified by the path, and returns the item number for the directory.

Arguments

path

An absolute or relative pathname for the new current directory.

Return Value

Returns the item number of the new directory or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

GetDirectory()

ChangeDirectoryInDir

Changes the current directory relative to another directory.

Synopsis

```
Item ChangeDirectoryInDir(Item dirItem, const char *path);
```

Description

Changes the current task's working directory to the absolute or relative location specified by the path, and returns the item number for the directory.

Arguments

dirItem
The directory relative to which the pathname should be interpreted.

path
An absolute or relative pathname for the new current directory.

Return Value

Returns the item number of the new directory or a negative error code for failure.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

GetDirectory()

CloseDirectory

Closes a directory.

Synopsis

```
void CloseDirectory(Directory *dir);
```

Description

This function closes a directory that was previously opened using `OpenDirectoryItem()` or `OpenDirectoryPath()`. All resources get released.

Arguments

`dir`

A pointer to the directory structure for the directory to close. This may be NULL in which case this function does nothing.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:directoryfunctions.h>, System.m2/Boot/filesystem

See Also

`OpenDirectoryItem()`, `OpenDirectoryPath()`, `ReadDirectory()`

CreateDirectory

Creates a directory.

Synopsis

```
Err CreateDirectory(const char *path);
```

Description

This function creates a new directory.

Arguments

path
The pathname for the new directory.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

DeleteDirectory()

CreateDirectoryInDir

Creates a directory relative to another directory.

Synopsis

```
Err CreateDirectoryInDir(Item ditItem, const char *path);
```

Description

This function creates a new directory.

Arguments

dirItem

The directory relative to which the pathname should be interpreted.

path

The absolute or relative pathname for the new directory.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

DeleteDirectory()

DeleteDirectory

Deletes a directory.

Synopsis

```
Err DeleteDirectory(const char *path);
```

Description

This function deletes an existing directory.

Arguments

path
The pathname for the directory to delete.

Return Value

Returns 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

CreateDirectory()

DeleteDirectoryInDir

Deletes a directory relative to another directory.

Synopsis

```
Err DeleteDirectoryInDir(Item dirItem, const char *path);
```

Description

This function deletes an existing directory.

Arguments

dirItem

The directory relative to which the pathname should be interpreted.

path

The absolute or relative pathname for the directory to delete.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

CreateDirectory()

GetDirectory

Gets the item number and pathname for the current directory.

Synopsis

```
Item GetDirectory(char *pathBuf, int32 pathBufLen);
```

Description

This function returns the item number of the calling task's current directory. If pathBuf is non-NULL, it must point to a buffer of writable memory whose length is given in pathBufLen; the absolute pathname of the current working directory is stored into this buffer.

Arguments

pathBuf

A pointer to a buffer in which to receive the absolute pathname for the current directory. If you do not want to get the pathname string, use NULL as the value of this argument.

pathBufLen

The size of the buffer pointed to by the pathBuf argument, in bytes, or zero if you don't provide a buffer.

Return Value

Returns the item number of the current directory.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

ChangeDirectory()

OpenDirectoryItem

Opens a directory specified by an item.

Synopsis

```
Directory *OpenDirectoryItem(Item openFileItem);
```

Description

This function opens a directory. It allocates a new directory structure, opens the directory, and prepares for a traversal of the contents of the directory. Unlike `OpenDirectoryPath()`, you specify the file for the directory by its item number rather than by its pathname.

Arguments

`openFileItem`

The item number of the open file to use for the directory. When you later call `CloseDirectory()`, this file item will automatically be freed for you, you do not need to call `CloseFile()`.

Return Value

The function returns a pointer to the directory structure that is created or NULL if an error occurs.

Implementation

Folio call implemented in File folio V20.

Caveats

When you are done scanning the directory and call `CloseDirectory()`, the item you gave to `OpenDirectoryItem()` will automatically be closed for you. In essence, when you call `OpenDirectoryItem()`, you are giving away the File item to the folio, which will dispose of it when the directory is closed.

Associated Files

<:file:directoryfunctions.h>, System.m2/Boot/filesystem

See Also

`OpenFile()`, `OpenFileInDir()`, `OpenDirectoryPath()`, `CloseDirectory()`

OpenDirectoryPath

Opens a directory specified by a pathname.

Synopsis

```
Directory *OpenDirectoryPath( const char *path )
```

Description

This function opens a directory. It allocates a new directory structure, opens the directory, and prepares for a traversal of the contents of the directory. Unlike `OpenDirectoryItem()`, you specify the file for the directory by its pathname rather than by its item number.

Arguments

`path`
An absolute or relative pathname for the file to scan.

Return Value

The function returns a pointer to the directory structure that is created or NULL if an error occurs.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:directoryfunctions.h>, System.m2/Boot/filesystem

See Also

`OpenFile()`, `OpenFileInDir()`, `OpenDirectoryItem()`, `CloseDirectory()`

ReadDirectory

Reads the next entry from a directory.

Synopsis

```
Err ReadDirectory(Directory *dir, DirectoryEntry *de);
```

Description

This routine reads the next entry from the specified directory. It stores the information from the directory entry into the supplied DirectoryEntry structure. You can then examine the DirectoryEntry structure for information about the entry.

The most interesting fields in the DirectoryEntry structure are:

de_FileName

The name of the entry.

de_Flags

This contains a series of bit flags that describe characteristics of the entry. Flags of interest are FILE_IS_DIRECTORY, which indicates the entry is a nested directory and FILE_IS_READONLY, which tells you the file cannot be written to.

de_Type

This is currently one of FILE_TYPE_DIRECTORY, FILE_TYPE_LABEL, or FILE_TYPE_CATAPULT.

de_BlockSize

This is the size in bytes of the blocks when reading this entry.

de_ByteCount

The logical count of the number of useful bytes within the blocks allocated for this file.

de_BlockCount

The number of blocks allocated for this file.

You can use OpenDirectoryPath() and ReadDirectory() to scan the list of mounted file systems. This is done by supplying a path of "/" to OpenDirectoryPath(). The entries that ReadDirectory() returns will correspond to all of the mounted file systems. You can then look at the de_Flags field to determine if a file system is readable or not.

Arguments

dir

A pointer to the directory structure for the directory.

de

A pointer to a DirectoryEntry structure in which to receive the information about the next directory entry.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. You will get a negative result when you reach the end of the directory.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:directoryfunctions.h>, System.m2/Boot/filesystem

See Also

OpenDirectoryItem(), OpenDirectoryPath()

CloseFile

Closes a file.

Synopsis

```
int32 CloseFile(Item fileItem);
```

Description

Closes a disk file that was opened with a call to `OpenFile()` or `OpenFileInDir()`. The specified item may not be used after successful completion of this call.

Arguments

`fileItem`

The item number of the disk file to close.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

`<:file:filefunctions.h>`, `System.m2/Boot/filesystem`

See Also

`OpenFile()`, `OpenFileInDir()`

CreateAlias

Creates a file system alias.

Synopsis

```
Item CreateAlias(const char *aliasPath, const char *realPath);
```

Description

This function creates an alias for a file. The alias can be used in place of the full pathname for the file. Note that the file system maintains separate alias entries for each task.

Arguments

aliasPath

The alias name.

realPath

The substitution string for the alias.

Return Value

Returns the item number for the file alias that is created, or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

OpenFile(), OpenFileInDir()

CreateFile

Creates a file.

Synopsis

```
Err CreateFile(const char *path);
```

Description

This function creates a new file.

Arguments

path
The pathname for the file.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

DeleteFile(), OpenFile(), OpenFileInDir()

CreateFileInDir

Creates a file relative to a directory.

Synopsis

```
Err CreateFileInDir(Item dirItem, const char *path);
```

Description

This function creates a new file.

Arguments

dirItem

The directory relative to which the pathname should be interpreted.

path

The absolute or relative pathname for the file.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

DeleteFile(), OpenFile(), OpenFileInDir()

DeleteFile

Deletes a file.

Synopsis

```
Err DeleteFile(const char *path);
```

Description

This function deletes a file.

Arguments

path
The pathname for the file to delete.

Return Value

Returns ≥ 0 if the file was successfully deleted or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

CreateFile(), OpenFile(), OpenFileInDir(), CloseFile()

DeleteFileInDir

Deletes a file relative to a directory.

Synopsis

```
Err DeleteFileInDir(Item dirItem, const char *path);
```

Description

This function deletes a file.

Arguments

dirItem

The directory relative to which the pathname should be interpreted.

path

The absolute or relative pathname for the file to delete.

Return Value

Returns ≥ 0 if the file was successfully deleted or a negative error code for failure.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

CreateFile(), OpenFile(), OpenFileInDir(), CloseFile()

FindFileAndIdentify

Searches one or more locations for a file, and returns its pathname

Synopsis

```
Err FindFileAndIdentify(char *pathBuf, int32 pathBufLen,  
                        const char *partialPath, TagArg *tags);  
  
Err FindFileAndIdentifyVA(char *pathBuf, int32 pathBufLen,  
                          const char *partialPath, uint32 tag, ...);
```

Description

This function searches one or more locations for a file. Tags are used to identify the locations to be searched, to require that the version and/or revision of the file be screened against a set of suitability criteria, and whether the search should scan all locations for the "most recent" version/revision of the file or should stop upon finding the first match.

Arguments

pathBuf
A buffer into which the call may return the complete (absolute) pathname of the file which was found.

pathBufLen
The size of the pathBuf buffer.

partialPath
A relative path, which should be interpreted relative to each of the search locations identified in the tag list.

tags
A tagarg array, where the tags specify version and revision matching criteria, how to handle files having no valid version and revision numbers, whether to stop on the first file found or scan exhaustively for the highest version, and the identity of the directories to be searched. See the FILESEARCH_TAG tags in *<:file:filesystem.h>*

Return Value

Returns ≥ 0 if the file is found and its name has been returned in pathBuf, or returns a negative error code for failure.

Implementation

Folio call implemented in File folio V29.

Associated Files

<:file:filesystem.h>, *<:file:filefunctions.h>*, System.m2/Boot/filesystem

See Also

FindFileAndOpen()

FindFileAndOpen

Searches one or more locations for a file, and opens the file.

Synopsis

```
Item FindFileAndOpen(const char *partialPath, TagArg *tags);  
  
Item FindFileAndOpenVA(const char *partialPath, uint32 tag, ...);
```

Description

This function searches one or more locations for a file. Tags are used to identify the locations to be searched, to require that the version and/or revision of the file be screened against a set of suitability criteria, and whether the search should scan all locations for the "most recent" version/revision of the file or should stop upon finding the first match.

Arguments

partialPath

A relative path, which should be interpreted relative to each of the search locations identified in the tag list.

tags

A tagarg array, where the tags specify version and revision matching criteria, how to handle files having no valid version and revision numbers, whether to stop on the first file found or scan exhaustively for the highest version, and the identity of the directories to be searched. See the FILESEARCH_TAG tags in <:file:filesystem.h>

Return Value

Returns the item number of the opened file (which can be used later to refer to the file), or a negative error code for failure.

Implementation

Folio call implemented in File folio V29.

Associated Files

<:file:filesystem.h>, <:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

FindFileAndIdentify()

OpenFile

Opens a disk file.

Synopsis

```
Item OpenFile(const char *path);
```

Description

This function opens a disk file, given an absolute or relative pathname, and returns its item number.

Arguments

path

An absolute or relative pathname for the file to open, or an alias for the pathname.

Return Value

Returns the item number of the opened file (which can be used later to refer to the file), or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

CloseFile(), OpenFileInDir()

OpenFileInDir

Opens a disk file relative to a directory.

Synopsis

```
Item OpenFileInDir(Item dirItem, const char *path);
```

Description

Similar to `OpenFile()`, this function allows the caller to specify the item of a directory that should serve as a starting location for the file search.

Arguments

dirItem
The item number of the directory containing the file.

path
A pathname for the file that is relative to the directory specified by the `dirItem` argument.

Return Value

Returns the item number of the opened file (which can be used later to refer to the file), or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

`OpenFile()`, `OpenDirectoryItem()`, `OpenDirectoryPath()`, `CloseFile()`

Rename

Rename a file, directory, or filesystem.

Synopsis

```
Err Rename(const char *path, const char *newName);
```

Description

Rename changes the name of a file, directory or filesystem. If the path is a file, the file name is replaced with newName. If the path is a directory, the directory name is replaced with newName. If the path is the root directory of the filesystem, then the name of the filesystem is replaced with newName. Currently, only acrobat filesystem allows this call.

Arguments

path

The path to the file or directory to be renamed.

newName

The name of the file or directory to change to. This must be a valid file name.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in File folio V20.

Associated Files

<:file:filefunctions.h>, System.m2/Boot/filesystem

See Also

DeleteFile(), OpenFile(), OpenFileInDir()

SetFileAttrs

Sets some attributes of a file.

Synopsis

```
Err SetFileAttrs(const char *path, const TagArg *tags);  
  
Err SetFileAttrsVA(const char *path, uint32 tag, ...);
```

Description

This function lets you set the value of fields associated with each file.

Arguments

path

The path name leading to the file to set the attributes of.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

FILEATTRS_TAG_FILETYPE (PackedID)

This tag lets you specify the 4-byte file type that is associated with every file.

FILEATTRS_TAG_VERSION (uint8)

Lets you specify the version associated with the file.

FILEATTRS_TAG_REVISION (uint8)

Lets you specify the revision associated with the file.

FILEATTRS_TAG_BLOCKSIZE (uint32)

Lets you specify the logical block size to use for this file. This tag can only be used if the file is currently empty.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

FILE_ERR_BADFILE

A bad file pointer was passed in.

FILE_ERR_BADSIZE

An attempt was made to set the block size while the file was not empty.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

ClearRawFileError

Clears the error state of a file.

Synopsis

```
Err ClearRawFileError(RawFile *file);
```

Description

After an error occurs while reading, writing, or seeking in a file, further operations to that file are automatically rejected and all return error codes. This makes it generally easy to deal with errors since it is only necessary to check the return status of the last I/O performed instead of all I/O calls.

In some cases, it is desirable not to have errors always be fatal. In particular, when an error condition can be corrected, it is useful to resume I/O operations where they left off. For example, if the error occurred because the user removed the media, it is desirable to prompt the user for the media, and resume the I/O operation as soon as the media becomes again mounted.

This function lets you clear the error state of an opened file. This enables I/O operations to the file once again.

Arguments

file

The file to clear the error state for.

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), ReadRawFile(), WriteRawFile(),
SeekRawFile(), GetRawFileInfo(), SetRawFileAttrs(), SetRawFileSize()

CloseRawFile

Concludes access to a file.

Synopsis

```
Err CloseRawFile(RawFile *file);
```

Description

This function concludes access to a file. It flushes any buffers that are dirty and releases all resources maintained for the file.

If a read, write, or seek operation failed while this file was opened, `CloseRawFile()` will return the error code associated with the failure, unless `ClearRawFileError()` was used to reset the error state.

Arguments

file

The file to close. This value may be NULL, in which case this function does nothing.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. If a read, write, or seek operation failed while this file was opened, the error code associated with this failure will be returned by this function, unless `ClearRawFileError()` is used to clear the error state.

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

`OpenRawFile()`, `ReadRawFile()`, `WriteRawFile()`, `SeekRawFile()`,
`GetRawFileInfo()`, `SetRawFileAttrs()`, `ClearRawFileError()`,
`SetRawFileSize()`

GetRawFileInfo

Gets some information about an opened file.

Synopsis

```
Err GetRawFileInfo(RawFile *file, FileInfo *info, uint32 infoSize);
```

Description

This function gets some information about an opened file. The information includes the size of the file in bytes and blocks, the underlying item being used for communicating with the file system, and more.

Arguments

file

The file to get information on.

info

A structure to hold the information.

infoSize

The size of the information structure. This should be sizeof(FileInfo).

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

FILE_ERR_BADSIZE

An illegal value was given for the infoSize argument. The parameter should be equal to sizeof(FileInfo).

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), ReadRawFile(), WriteRawFile(),
SeekRawFile(), SetRawFileAttrs(), ClearRawFileError(), SetRawFileSize()

OpenRawFile

Gains access to a file for raw file I/O.

Synopsis

```
Err OpenRawFile(RawFile **file, const char *path, FileOpenModes
mode);
```

Description

This function does the necessary work to establish a connection to a file in a file system and prepare it for use. Once a file is opened, it can be read or written to.

Arguments

file

A pointer to a variable where a handle to the RawFile will be stored. The value is set to NULL if the file can't be opened.

path

The pathname leading to the file to be opened. This may be an absolute path, or relative to the current directory.

mode

The mode to open the file in. This determines what can be done to the file, and allows I/O operations to be optimized for the particular type of access mode.

The modes supported are:

FILEOPEN_READ

The file is being opened for reading only. Attempts to write to the file will fail. If the file doesn't exist, the open call will fail.

FILEOPEN_WRITE

The file is being opened for writing only. Attempts to read from the file will fail. If the file doesn't exist, it will be created. If the file already exists, the previous contents are left intact, and the file cursor is positioned at the beginning of the file.

FILEOPEN_WRITE_NEW

The file is being opened for writing only. Attempts to read from the file will fail. If the file doesn't exist, it will be created. If the file already exists, it is first deleted, and then recreated, which erases any previous contents.

FILEOPEN_WRITE_EXISTING

The file is being opened for writing only. Attempts to read from the file will fail. If the file doesn't exist, the open call will fail. If the file already exists, the previous contents are left intact, and the file cursor is positioned at the beginning of the file.

FILEOPEN_READWRITE

Same as FILEOPEN_WRITE, except that the file can also be read.

FILEOPEN_READWRITE_NEW

Same as FILEOPEN_WRITE_NEW, except that the file can also be read.

FILEOPEN_READWRITE_EXISTING

Same as FILEOPEN_WRITE_EXISTING, except that the file can also be read.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

`FILE_ERR_NOFILE`

No file of the specified name could be found.

`FILE_ERR_NOMEM`

There was not enough memory to allocate the resources needed.

`FILE_ERR_BADMODE`

An illegal open mode was supplied.

Implementation

Folio call implemented in File folio V27.

Associated Files

`<:file:fileio.h>`, `<:file:filesystem.h>`, `System.m2/Boot/filesystem`

See Also

`CloseRawFile()`, `ReadRawFile()`, `WriteRawFile()`, `SeekRawFile()`,
`GetRawFileInfo()`, `SetRawFileAttrs()`, `ClearRawFileError()`,
`SetRawFileSize()`, `OpenRawFileInDir()`

OpenRawFileInDir()

Gains access to a file for raw file I/O.

Synopsis

```
Err OpenRawFileInDir(RawFile **file, Item dirItem, const char *path,  
                    FileOpenModes mode);
```

Description

This function does the necessary work to establish a connection to a file in a file system and prepare it for use. Once a file is opened, it can be read or written to.

Arguments

file

A pointer to a variable where a handle to the RawFile will be stored. The value is set to NULL if the file can't be opened.

dirItem

The item number of the directory containing the file.

path

A pathname for the file that is relative to the directory specified by the dirItem argument.

mode

The mode to open the file in. This determines what can be done to the file, and allows I/O operations to be optimized for the particular type of access mode.

The modes supported are:

FILEOPEN_READ

The file is being opened for reading only. Attempts to write to the file will fail. If the file doesn't exist, the open call will fail.

FILEOPEN_WRITE

The file is being opened for writing only. Attempts to read from the file will fail. If the file doesn't exist, it will be created. If the file already exists, the previous contents are left intact, and the file cursor is positioned at the beginning of the file.

FILEOPEN_WRITE_NEW

The file is being opened for writing only. Attempts to read from the file will fail. If the file doesn't exist, it will be created. If the file already exists, it is first deleted, and then recreated, which erases any previous contents.

FILEOPEN_WRITE_EXISTING

The file is being opened for writing only. Attempts to read from the file will fail. If the file doesn't exist, the open call will fail. If the file already exists, the previous contents are left intact, and the file cursor is positioned at the beginning of the file.

FILEOPEN_READWRITE

Same as FILEOPEN_WRITE, except that the file can also be read.

FILEOPEN_READWRITE_NEW

Same as FILEOPEN_WRITE_NEW, except that the file can also be read.

FILEOPEN_READWRITE_EXISTING

Same as FILEOPEN_WRITE_EXISTING, except that the file can also be read.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

FILE_ERR_NOFILE

No file of the specified name could be found.

FILE_ERR_NOMEM

There was not enough memory to allocate the resources needed.

FILE_ERR_BADMODE

An illegal open mode was supplied.

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

CloseRawFile(), ReadRawFile(), WriteRawFile(), SeekRawFile(),
GetRawFileInfo(), SetRawFileAttrs(), ClearRawFileError(),
SetRawFileSize(), OpenRawFile()

ReadRawFile

Reads data from a file.

Synopsis

```
int32 ReadRawFile(RawFile *file, void *buffer, int32 numBytes);
```

Description

This function reads data from a file starting at the current file cursor position. The data is copied into the supplied buffer. Once the data is read, the file cursor is advanced by the number of bytes read. This causes sequential read operations to progress through all the data in the file.

If there is an error while reading the file, an error code will be returned, and the contents of the supplied buffer will be undefined. Once an error occurs for a file, all subsequent operations to that file are blocked and will all return errors. When you finally close the file, `CloseRawFile()` will return the error code which describes the failure. You can use `ClearRawFileError()` to reset the error state and once again allow I/O operations to be performed to the file.

When reading a file, it is slightly more efficient to supply a read buffer which is an integral multiple in size of the file's block size. You can obtain the file's block size using the `GetRawFileInfo()` function.

Arguments

file

The file to read from.

buffer

A memory buffer where the data is to be copied.

numBytes

The number of bytes to read from the file. If there aren't that many bytes left, the maximum number of bytes possible will be read.

Return Value

Returns the number of bytes read into the buffer, or a negative error code for failure. When a failure occurs, the contents of the supplied buffer are undefined.

Upon failure, the file cursor will be left in its original position, so that if the error condition is corrected, it is simply a matter of calling `ReadRawFile()` again with the same data. For example, if the failure occurred because the media was removed, when the user reinserts the media, it is possible to resume the reading operations right where it left off.

Possible error codes currently include:

`FILE_ERR_BADFILE`

A bad file pointer was passed in.

`FILE_ERR_BADCOUNT`

A negative value was supplied for the numBytes argument.

`FILE_ERR_OFFLINE`

The file system is no longer mounted (the user likely removed the media).

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), WriteRawFile(), SeekRawFile(),
GetRawFileInfo(), SetRawFileAttrs(), ClearRawFileError(),
SetRawFileSize()

SeekRawFile

Moves the file cursor within a file.

Synopsis

```
int32 SeekRawFile(RawFile *file, int32 position, FileSeekModes mode);
```

Description

This function moves the file cursor within the file. The file cursor determines where the next read operation will get its data from, and where the next write operation will write its data to.

The file cursor can be moved to a position which is relative to the start or end of file, as well as relative to the current cursor position. The file cursor is never allowed to be less than 0 or greater than the number of bytes in the file.

Arguments

file

The file to affect.

position

The position to move the file cursor to.

mode

Describes what the position argument is relative to.

The possible seek modes are:

FILESEEK_START

Indicates that the supplied position is relative to the beginning of the file. So a position of 10 would put the file cursor at byte #10 (the eleventh byte) within the file.

FILESEEK_CURRENT

Indicates that the supplied position is relative to the current position within the file. A positive position value moves the cursor forward in the file by that many bytes, while a negative value moves the cursor back by that number of bytes.

FILESEEK_END

Indicates that the supplied position is relative to the end of the file. The position value should be a negative value.

Return Value

Returns the previous position of the file cursor within the file or a negative error code for failure. Upon failure, the file cursor will not have been moved. Trying to seek beyond the bounds of the file is not considered an error and is handled by limiting the motion of the file cursor within allowed bounds.

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), ReadRawFile(), WriteRawFile(),
GetRawFileInfo(), SetRawFileAttrs(), ClearRawFileError(),
SetRawFileSize()

SetRawFileAttrs

Sets some attributes of an opened file.

Synopsis

```
Err SetRawFileAttrs(RawFile *file, const TagArg *tags);  
  
Err SetRawFileAttrsVA(RawFile *file, uint32 tag, ...);
```

Description

This function lets you set the value of fields associated with each file. You can only call this function if the file was opened in a writable mode.

Arguments

file
The file to set the attributes of.

tags
A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

FILEATTRS_TAG_FILETYPE (PackedID)

This tag lets you specify the 4-byte file type that is associated with every file.

FILEATTRS_TAG_VERSION (uint8)

Lets you specify the version associated with the file.

FILEATTRS_TAG_REVISION (uint8)

Lets you specify the revision associated with the file.

FILEATTRS_TAG_BLOCKSIZE (uint32)

Lets you specify the logical block size to use for this file. This tag can only be used if the file is currently empty.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

FILE_ERR_BADFILE

A bad file pointer was passed in.

FILE_ERR_BADMODE

The file was not in a valid mode to call this function.

FILE_ERR_BADSIZE

An attempt was made to set the block size while the file was not empty.

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), ReadRawFile(), WriteRawFile(),
SeekRawFile(), GetRawFileInfo(), ClearRawFileError(), SetRawFileSize()

SetRawFileSize

Sets the size of an opened file.

Synopsis

```
Err SetRawFileSize(RawFile *file, uint32 newSize);
```

Description

This function lets you set the size of a file on disk.

Arguments

file

The file to set the size of.

newSize

The new size in bytes of the file. If the file is being made smaller, the file cursor is automatically moved, if needed, to not exceed the new end of the file.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

FILE_ERR_BADFILE

A bad file pointer was passed in.

FILE_ERR_BADMODE

The file was not in a valid mode to call this function.

Implementation

Folio call implemented in File folio V28.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), ReadRawFile(), WriteRawFile(),
SeekRawFile(), GetRawFileInfo(), ClearRawFileError(), SetRawFileAttrs()

WriteRawFile

Writes data to a file.

Synopsis

```
int32 WriteRawFile(RawFile *file, const void *buffer, int32
numBytes);
```

Description

This function writes data to a file starting at the current file cursor position. The data is copied from the supplied buffer. Once the data is written, the file cursor is advanced by the number of bytes written. This causes sequential write operations to progress forward and extend the file as new data is written.

If there is an error while writing the file, an error code will be returned, and whether any part of the data has made it to the file is undefined. Once an error occurs for a file, all subsequent operations to that file are blocked and will all return errors. When you finally close the file, `CloseRawFile()` will return the error code which describes the failure. You can use `ClearRawFileError()` to reset the error state and once again allow I/O operations to be performed to the file.

As data is written at the end of a file, the size of the file is automatically increased to hold the new data. If you know the amount of data you will be writing to a file ahead of time, it is more efficient to use the `SetRawFileSize()` function to preallocate room on the media for the file. This helps reduce media fragmentation and improve overall performance.

Arguments

`file`

The file to write to.

`buffer`

A memory buffer containing the data to write.

`numBytes`

The number of bytes to write to the file.

Return Value

Returns the number of bytes written to the file, or a negative error code for failure. When a failure occurs, parts of the data may have already been written to the file.

Upon failure, the file cursor will be left in its original position, so that if the error condition is corrected, it is simply a matter of calling `WriteRawFile()` again with the same data. For example, if the failure occurred because the media was full, it is possible to just delete some files, and resume the writing operations right where it left off.

Possible error codes currently include:

`FILE_ERR_BADFILE`

A bad file pointer was passed in.

`FILE_ERR_BADCOUNT`

A negative value was supplied for the `numBytes` argument.

`FILE_ERR_NOSPACE`

There's no more room on the media.

Implementation

Folio call implemented in File folio V27.

Associated Files

<:file:fileio.h>, <:file:filesystem.h>, System.m2/Boot/filesystem

See Also

OpenRawFile(), CloseRawFile(), ReadRawFile(), SeekRawFile(),
GetRawFileInfo(), SetRawFileAttrs(), ClearRawFileError(),
SetRawFileSize()

Chapter 9

FileSystem Utilities Folio Calls

This section presents the reference documentation for the FSUtils folio, which provides high-level functions to perform various file system operations.

AppendPath

Appends a path to an existing path specification.

Synopsis

```
Err AppendPath(char *path, const char *append,  
               uint32 numBytes);
```

Description

This function appends a file, directory, or subpath name to an existing path specification, dealing with slashes and absolute vs relative directory paths.

Arguments

path

The path to append to.

append

The file name, directory name, or full path to append.

numBytes

The number of bytes available in the resulting path buffer.

Return Value

Returns the number of bytes in the resulting path, or -1 if the path buffer wasn't big enough to receive the final path.

Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<:file:fsutils.h>, System.m2/Modules/fsutils

See Also

FindFinalComponent()

CreateCopyObj

Creates a copier object to perform hierarchical copy operations.

Synopsis

```
Err CreateCopyObj(CopyObj **co, const char *sourceDir,  
                 const char *objName, const TagArg *tags);  
  
Err CreateCopyObjVA(CopyObj **co, const char *sourceDir,  
                   const char *objName, uint32 tag, ...);
```

Description

This function creates a file copier object. The object lets you perform file copy operations of full directory hierarchies between any two directories or devices available.

The copier object involves a set of state machines and algorithms to minimize the number of media swaps the user has to perform when doing copies from one card to the other on a system with a single card slot.

The copier object provides no user-interface. Whenever user interaction is required, callback hooks are invoked. The user of the copy object is then free to bring up any UI component needed to effect the desired interaction.

For all callback functions, a return value of 0 indicates success, while all other values are considered requests to stop the copy process. In such a case, the value returned by the callback function is used as return value from the copier function itself. For most callbacks, a return value of 0 tells the copier to retry the operation.

The copier operates in a non-recursive manner, which prevents running out of stack space when copying a deep directory structure.

Arguments

co

A pointer to a variable where a handle to the copier object will be stored. The value is set to NULL if the object can't be opened.

sourceDir

A path specification denoting the location in the file system where the object to copy is located. This path can be absolute or relative.

objName

The name of the file system object to copy. This is either the name of a file or a directory within the directory specified by the sourceDir argument.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

COPIER_TAG_USERDATA (void *)

A value that the copier object will pass to all callback functions that it calls. You can use this to supply a pointer to your context data. If you don't supply this tag, NULL will be passed to the callbacks.

COPIER_TAG_MEMORYTHRESHOLD (uint32)

This tag lets you specify the maximum amount of memory that the copier object will allocate. In general, this amount will be exceeded slightly. If this tag is not provided, the default is set to 1M.

COPIER_TAG_ASYNCHRONOUS (bool)

Specifies whether the reading part of the copier should operate asynchronously. This allows read and write operations to overlap, which may effectively cut the time to do the data transfer in half. More memory is needed when running asynchronously, mainly used for a thread's stack. The default is to run the copy synchronously.

COPIER_TAG_PROGRESSFUNC (CopyProgressFunc)

This tag lets you supply a callback function that is called during the copy process to let you update a UI display. The callback is supplied the current file name being worked on.

COPIER_TAG_NEEDSOURCEFUNC (CopyNeedSourceFunc)

This tag lets you supply a callback function that is called during the copy process whenever the copier needs to see the source media. It lets you bring up a UI to prompt the user to insert the media. The callback is provided the name of the file system which is sought. If this tag is not provided and the source media can't be found, the copy operation will stop with a FILE_ERR_OFFLINE error.

COPIER_TAG_NEEDDESTINATIONFUNC (CopyNeedDestinationFunc)

This tag lets you supply a callback function that is called during the copy process whenever the copier needs to see the destination media. It lets you bring up a UI to prompt the user to insert the media. The callback is provided the name of the file system which is sought. If this tag is not provided and the destination media can't be found, the copy operation will stop with a FILE_ERR_OFFLINE error.

COPIER_TAG_DUPLICATEFUNC (CopyDuplicateFunc)

This tag lets you supply a callback function that is called during the copy process whenever an object of the same name as objName already exists in the destination directory. The callback function can delete this duplicate object, or rename it, or do nothing at all. If the callback returns 0, the copy operation will continue. If the duplicate not renamed or deleted, the new object will overwrite the existing object. In the case of directories, a merge operation will effectively be done.

COPIER_TAG_DESTINATIONFULLFUNC (CopyDestinationFullFunc)

This tag lets you supply a callback function that is called during the copy process whenever the copier determines that there isn't enough room on the destination media. It lets you bring up a UI to inform the user of this fact. The callback could proceed to make more room on the destination by deleting some files, or it may just return FILE_ERR_NOSPACE to stop the copy operation.

COPIER_TAG_DESTINATIONPROTECTEDFUNC (CopyDestinationProtectedFunc)

This tag lets you supply a callback function that is called during the copy process whenever the copier determines that the destination media is write-protected. It lets you bring up a UI to inform the user of this fact. The callback can return 0 to have the operation retried, or return FILE_ERR_READONLY to stop the copy operation.

COPIER_TAG_READERRORFUNC (CopyReadErrorFunc)

This tag lets you supply a callback function that is called during the copy process whenever an unexpected error occurs while reading the source media. The callback is supplied the error code. If the callback can do anything to rectify the problem, it may return 0 to have the operation retried, or it may simply return the supplied error code to stop the copy operation.

COPIER_TAG_WRITEERRORFUN (CopyWriteErrorFunc)

This tag lets you supply a callback function that is called during the copy process whenever an unexpected error occurs while writing to the destination media. The callback is supplied the error code. If the callback can do anything to rectify the problem, it may return 0 to have the operation retried, or it may simply return the supplied error code to stop the copy operation.

Return Value

Returns ≥ 0 if the object was created successfully, or a negative error code for failure.

Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<.:file:fsutils.h>, System.m2/Modules/fsutils

See Also

DeleteCopyObj(), PerformCopy()

DeleteCopyObj

Releases any resources consumed by
CreateCopyObj()

Synopsis

```
Err DeleteCopyObj(CopyObj *co);
```

Description

This function releases any resources consumed by CreateCopyObj(). This might involved freeing memory, flushing I/O buffers, closing files and more.

Callbacks hooks supplied when the object was created may be called as this function executes.

Arguments

co

The copy object obtained from CreateCopyObj(), or NULL in which case this function does nothing.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

- Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<:file:fsutils.h>, System.m2/Modules/fsutils

See Also

CreateCopyObj(), PerformCopy()

DeleteTree

Deletes a complete filesystem directory tree.

Synopsis

```
Err DeleteTree(const char *path, const TagArg *tags);  
Err DeleteTreeVA(const char *path, uint32 tag, ...);
```

Description

This function lets you delete an entire directory tree. The deletion is done without recursion, which avoids running out of stack space on deep directories.

The deleter engine provides no user-interface. Whenever user interaction is required, callback hooks are invoked. The user of the deleter engine is then free to bring up any UI component needed to effect the desired interaction.

The callbacks are invoked with different types of information, depending on the callback. The functions can return 0 to tell the deleter engine to retry the operation, 1 to tell the deleter to skip the current file or directory and move on, or any other value to cause the deleter to exit.

Arguments

path

The file system component to delete. This can be a file or a directory. If it is a directory, an attempt will made to delete all nested objects before deleting the directory itself.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

DELETER_TAG_USERDATA (void *)

A value that the deleter engine will pass to all callback functions that it calls. You can use this to supply a pointer to your context data. If you don't supply this tag, NULL will be passed to the callbacks.

DELETER_TAG_PROGRESSFUNC (DeleteProgressFunc)

This tag lets you supply a callback function that is called during the deletion process to let you update a UI display. The callback is supplied the current file name being deleted.

DELETER_TAG_NEEDMEDIAFUNC (DeleteNeedMediaFunc)

This tag lets you supply a callback function that is called during the deletion process whenever the deleter needs to see the media. It lets you bring up a UI to prompt the user to insert the media. The callback is provided the name of the file system which is sought. If this tag is not provided and the media can't be found, the copy operation will stop with a FILE_ERR_OFFLINE error.

DELETER_TAG_MEDIAPROTECTEDFUNC (DeleteMediaProtectedFunc)

This tag lets you supply a callback function that is called during the deletion process whenever the deleter determines that the media is write-protected. It lets you bring up a UI to inform the user of this fact. The callback can return 0 to have the operation retried, or return FILE_ERR_READONLY to stop the deletion operation.

DELETER_TAG_ERRORFUNC (DeleteErrorFunc)

This tag lets you supply a callback function that is called during the deletion process whenever an unexpected error occurs accessing the media. The callback is supplied the error code. If the callback can do anything to rectify the problem, it may return 0 to have the operation retried, or it may simply return the supplied error code to stop the deletion operation.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Upon failure, files might have already been deleted from the directory tree.

Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<:file:fsutils.h>, System.m2/Modules/fsutils

See Also

DeleteFile(), DeleteDirectory()

FindFinalComponent

Returns the last component of a path.

Synopsis

```
char *FindFinalComponent(const char *path);
```

Description

This function returns a pointer to the last component of a path specification. If there is only one component in the path, it returns a pointer to the beginning of the path string.

Arguments

path
The path specification.

Return Value

Returns a pointer to the last component of the path.

Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<:file:fsutils.h>, System.m2/Modules/fsutils

See Also

AppendPath()

GetPath

Obtain the absolute directory path leading to an opened file.

Synopsis

```
Err GetPath(Item file, char *path, uint32 numBytes);
```

Description

This function does the needed legwork to obtain the full path leading to an already opened file item. You can achieve the same thing on your own by sending a FILECMD_GETPATH IOREq to the file.

Arguments

file

A file item as obtained from the File folio's `OpenFile()` function.

path

A pointer to a memory buffer to hold the resulting path name.

numBytes

The number of bytes available in the path buffer.

Return Value

Returns the number of bytes in the resulting path, or a negative error code for failure.

Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<:file:fsutils.h>, System.m2/Modules/fsutils

See Also

`AppendPath()`, `FindFinalComponent()`

PerformCopy

Enter the copier engine and start copying files.

Synopsis

```
Err PerformCopy(CopyObj *co, const char *destinationDir);
```

Description

This function is the entry point into the copier engine. It tells the copy object where its data should be put. As this function executes, any callbacks hooks you supplied when creating the copy object may be invoked.

Arguments

co

The copy object, as obtained from `CreateCopyObj()`.

destinationDir

The destination directory where the object being copied should be put. This can be an absolute or relative path which may or may not be on the same physical media as the object being copied.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Upon failure, it is possible that some files or directories might have already been copied to the destination.

Implementation

Folio call implemented in FSUtils Folio V28.

Associated Files

<:file:fsutils.h>, System.m2/Modules/fsutils

See Also

`CreateCopyObj()`, `DeleteCopyObj()`

Chapter 10

Icon Folio Calls

This section presents the reference documentation for the Icon folio, which provides icon management functions.

LoadIcon

Loads icon data into memory and prepares it for rendering.

Synopsis

```
Err LoadIcon(Icon **icon, const TagArg *tags);  
Err LoadIconVA(Icon **icon, uint32 tag, ...);
```

Description

This function provides a method for retrieving icons from many different parts of the system, including from a file, a file system, a type of hardware, or a device.

Arguments

icon

A pointer to a variable where a handle to the Icon will be stored. The value is set to NULL if the icon can't be loaded.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

LOADICON_TAG_FILENAME (char *)

Defines the file path of an icon file to load. This tag is mutually exclusive with LOADICON_TAG_IFFPARSER, LOADICON_TAG_FILESYSTEM, LOADICON_TAG_DRIVER, and LOADICON_TAG_HARDWARE.

LOADICON_TAG_IFFPARSER (IFFParser *)

This tag permits the caller to utilize an already existing IFFParser structure from the IFF folio. This is useful for embedding icon data in other IFF files. This tag is mutually exclusive with LOADICON_TAG_FILENAME, LOADICON_TAG_FILESYSTEM, LOADICON_TAG_DRIVER, and LOADICON_TAG_HARDWARE. Note that use of this tag requires the presence of LOADICON_TAG_IFFPARSETYPE to be functional.

LOADICON_TAG_IFFPARSETYPE (uint32)

This tag defines options for IFF parsing, and is only valid when present with the LOADTEXTURE_TAG_IFFPARSER tag. The following options are available, and can be OR'd together:

LOADICON_TYPE_AUTOPARSE Requests that LoadIcon() parse for a FORM ICON chunk.

LOADICON_TYPE_PARSED Indicates that the caller has already parsed up to a FORM ICON, and that the current context of the IFF stream is within that chunk.

LOADICON_TAG_FILESYSTEM (char *)

When provided with a filesystem-style path, this tag will attempt to locate an icon associated with that filesystem. This tag is mutually exclusive with LOADICON_TAG_FILENAME, LOADICON_TAG_IFFPARSER, LOADICON_TAG_DRIVER, and LOADICON_TAG_HARDWARE.

LOADICON_TAG_DRIVER (char *)

When provided with the name of a device driver, this tag will attempt to locate an icon associated with that device driver without loading the driver itself into memory. This tag is mutually exclusive with LOADICON_TAG_FILENAME, LOADICON_TAG_IFFPARSER, LOADICON_TAG_FILESYSTEM, and LOADICON_TAG_HARDWARE.

LOADICON_TAG_HARDWARE (HWId)

This tag permits the caller to request an icon for a given hardware entity. This tag is mutually exclusive with LOADICON_TAG_FILENAME, LOADICON_TAG_IFFPARSER, LOADICON_TAG_FILESYSTEM, and LOADICON_TAG_DRIVER.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Icon folio V30.

Associated Files

<:ui:icon.h>

See Also

UnloadIcon()

SaveIcon

Creates an IFF Icon file.

Synopsis

```
Err SaveIcon(char *UTFfilename, char *AppName, const TagArg *tags);  
Err SaveIconVA(char *UTFfilename, char *AppName, uint32 tag, ...);
```

Description

This function permits the caller to create either a stand-alone Icon file, or a FORM ICON chunk within another IFF file.

Arguments

UTFfilename

Defines the name of an existing UTF file which contains the texture data to be used for this icon. Either single or multiple-texture UTF files are appropriate.

AppName

A string (which is no greater in length than 31 characters) that contains a user-readable string that describes the name of the application that created the icon. (Such as "SuperShark III".)

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

SAVEICON_TAG_FILENAME (char *)

Defines the file path to use when creating the icon file. This tag is mutually exclusive with SAVEICON_TAG_IFFPARSER.

SAVEICON_TAG_IFFPARSER (IFFParser *)

This tag, which is mutually exclusive with SAVEICON_TAG_FILENAME, permits the caller to write out a full IFF Icon data chunk (a FORM ICON) to an existing IFFParser *, from the IFF Folio. This permits Icons to be imbedded in other IFF files.

SAVEICON_TAG_TIMEBETWEENFRAMES (TimeVal *)

When provided, this tag defines the amount of time that should be waited when displaying each frame of a multiple-textured Icon. If not supplied, time of 0 seconds, 0 microseconds, will be saved to the icon.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Icon folio V30.

Associated Files

<:ui:icon.h>

See Also

LoadIcon(), UnloadIcon()

UnloadIcon

Unloads an icon from memory and frees any resources associated with it.

Synopsis

```
Err UnloadIcon(Icon *icon);
```

Description

This function frees any resources allocated by `LoadIcon()`. After this call is made, the icon pointer becomes invalid.

Arguments

`icon`

Pointer to an Icon as obtained from `LoadIcon()`. This pointer may be NULL in which case this function does nothing.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Icon folio V30.

Associated Files

`<:ui:icon.h>`

See Also

`LoadIcon()`

Chapter 11

IFF Folio Calls

This section presents the reference documentation for the IFF folio and associated link libraries, which provides services to read and write files in the IFF format.

AllocContextInfo

Allocates and initializes a ContextInfo structure.

Synopsis

```
ContextInfo *AllocContextInfo(PackedID type, PackedID id,  
                              PackedID ident, uint32 dataSize,  
                              IFFCallback cb);
```

Description

Allocates and initializes a ContextInfo structure with a specified number of bytes of user data. The returned structure contains a pointer to the user data buffer in the ci_Data field. The buffer is automatically cleared to 0 upon allocation.

The optional callback function argument sets a client-supplied cleanup routine for disposal when the context associated with the new ContextInfo is popped. The purge routine will be called when the ContextNode containing this structure is popped off the context stack and is about to be deleted itself.

The purge callback you supply is called with its second argument as a pointer to the ContextInfo structure being purged. The purge routine is responsible for calling FreeContextInfo() on this ContextInfo when it is done with it.

Arguments

- | | |
|----------|---|
| type | ID of container (eg: AIFF) |
| id | ID of context that will contain this node. |
| ident | Identifier that represents the type of information this node describes. This value is later used when searching through the list of ContextInfo structures to find a particular node. |
| dataSize | The number of bytes of data to allocate for user use. The data area allocated can be found by looking at the ci_Data field of the returned ContextInfo structure. |
| cb | The routine to be called when the ContextInfo structure needs to be purged. If this is NULL, the folio will simply call FreeContextInfo() automatically. |

Return Value

Returns a pointer to a new ContextInfo structure or NULL if there was not enough memory.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

FreeContextInfo(), StoreContextInfo(), AttachContextInfo(),
RemoveContextInfo(), FindContextInfo()

AttachContextInfo

Attaches a ContextInfo structure to a given ContextNode.

Synopsis

```
void AttachContextInfo(IFFParser *iff, ContextNode *to,  
                      ContextInfo *ci);
```

Description

This function adds the ContextInfo structure to the supplied ContextNode structure. If another ContextInfo of the same type, id and ident codes is already present in the ContextNode, it will be purged and replaced with this new structure.

Arguments

iff
 The parsing handle to affect.

to
 The ContextNode to attach the ContextInfo structure to.

ci
 The ContextInfo structure to attach.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

StoreContextInfo(), RemoveContextInfo(), AllocContextInfo(),
FreeContextInfo(), FindContextInfo()

CreateIFFParser

Creates a new parser structure and prepares it for use.

Synopsis

```
Err CreateIFFParser(IFFParser **iff, bool writeMode,  
                    const TagArg tags[]);  
  
Err CreateIFFParserVA(IFFParser **iff, bool writeMode,  
                      uint32 tag, ...);
```

Description

This functions allocates and initializes a new IFFParser structure. Once created, you can use the structure to parse IFF streams and get at the data in them. The streams can currently either be files or a block of memory.

Arguments

iff
A pointer to a variable where a handle to the IFFParser will be stored. The value is set to NULL if the parser can't be created.

writeMode
Specifies how the stream should be opened. TRUE means the file will be written to, or FALSE if the stream is to be read.

tags
A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

IFF_TAG_FILE (const char *)

This tag lets you specify the name of a file to parse. This file will be opened and prepared for use automatically. If you supply this tag, you may not supply the IFF_TAG_IOFUNCS tag at the same time.

IFF_TAG_IOFUNCS (IFFIOFuncs *)

This tag is used in conjunction with the IFF_TAG_IOFUNCS_DATA tag and lets you provide custom I/O functions that are used as callbacks by the folio to read and write data. This lets you do things like parse an in-memory IFF image, or parse a data stream coming from some place other than a file. If you supply this tag, you may not supply the IFF_TAG_FILE tag at the same time. You supply a pointer to a fully initialized IFFIOFuncs structure, which contains function pointers that the folio can use to perform I/O operations.

IFF_TAG_IOFUNCS_DATA (void *)

This tag works in conjunction with the IFF_TAG_IOFUNCS tag and lets you specify the parameter that is passed as the openKey parameter to the IFFOpenFunc callback function.

Return Value

Returns ≥ 0 for success, or a negative error code if the parser could not be created. Possible error codes currently include:

IFF_ERR_NOOPENTYPE

You didn't supply one of the IFF_TAG_FILE or IFF_TAG_IOFUNCS tags. One of these two tags must be supplied in order to tell the parser what data to parse.

IFF_ERR_NOMEM

There was not enough memory.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

DeleteIFFParser(), ParseIFF()

DeleteIFFParser

Deletes an IFF parser.

Synopsis

```
Err DeleteIFFParser(IFFParser *iff);
```

Description

Deletes an IFF parser previously created by `CreateIFFParser()`. Any data left to be output is sent out, files are closed, and all resources are released.

Arguments

`iff`

An active parser, as obtained from `CreateIFFParser()`. Once this call is made, the parser handle becomes invalid and can no longer be used. This value may be NULL, in which case this function does nothing.

Return Value

Returns ≥ 0 if successful, or a negative error code if it fails. Possible error codes currently include:

`IFF_ERR_BADPTR`

An invalid parser handle was supplied.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

`<:misc:iff.h>`, `System.m2/Modules/iff`

See Also

`CreateIFFParser()`, `ParseIFF()`

FindCollection

Gets a pointer to the current list of collection chunks.

Synopsis

```
CollectionChunk *FindCollection(const IFFParser *iff,  
                                PackedID type, PackedID id);
```

Description

This function returns a pointer to a list of `CollectionChunk` structures for each of the collection chunks of the given type encountered so far in the course of the parse operation. The structures appearing first in the list will be the ones encountered most recently.

Arguments

`iff`
The parsing handle to query.

`type`
Chunk type to search for.

`id`
Chunk id to search for.

Return Value

A pointer to the last `CollectionChunk` structure describing the last collection chunk encountered. The structure contains a pointer that goes back through all the collection chunks encountered during the parse operation. This function returns `NULL` if no collection chunks have been encountered so far.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

`RegisterCollectionChunks()`

FindContextInfo

Returns a ContextInfo structure from the context stack.

Synopsis

```
ContextInfo *FindContextInfo(const IFFParser *iff, PackedID type,  
                             PackedID id, PackedID ident);
```

Description

This functions searches through the context stack for a ContextInfo structure with matching type, id, and ident values. The search starts from the most current context backwards, so that the any structure found will be the one with greatest precedence in the context stack. The type, id, and ident values correspond to the parameters supplied to AllocContextInfo() to create the structure.

Arguments

iff	The parse handle to search in.
type	The type value to search for.
id	The id value to search for.
ident	The ident code to search for.

Return Value

Returns a pointer to a ContextInfo structure or NULL if no matching structure was found.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

AllocContextInfo(), FreeContextInfo(), StoreContextInfo(),
AttachContextInfo(), RemoveContextInfo()

FindPropChunk

Searches for a stored property chunk.

Synopsis

```
PropChunk *FindPropChunk(const IFFParser *iff,  
                          PackedID type, PackedID id);
```

Description

This function searches for the stored property chunk which is valid in the given context. Property chunks are automatically stored by `ParseIFF()` when pre-declared by `RegisterPropChunks()`. The `pc_Data` field of the `PropChunk` structure contains a pointer to the data associated with the property chunk.

The data pointed to by `pc_Data` is fully writable by your task. It can sometimes be handy to write directly to this area instead of making a local copy.

Arguments

- `iff`
The parser handler to search in.
- `type`
The type code of the property chunk to search for.
- `id`
The id code of the property chunk to search for.

Return Value

Returns a pointer to a `PropChunk` structure that describes the sought property, or `NULL` if no such property chunk has been encountered so far.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

`RegisterPropChunks()`

FindPropContext

Gets the parser's current property context.

Synopsis

```
ContextNode *FindPropContext(const IFFParser *iff);
```

Description

This function locates the context node which would be the scoping chunk for properties in the current parsing state. This is used for locating the proper scoping context for property chunks (i.e. the scope from which a property would apply). This is usually the FORM or LIST with the highest precedence in the context stack.

Arguments

iff
The parsing handle to search in.

Return Value

Returns a pointer to a ContextNode or NULL if not found.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

GetCurrentContext(), GetParentContext()

FreeContextInfo

Deallocate a ContextInfo structure.

Synopsis

```
void FreeContextInfo(ContextInfo *ci);
```

Description

This function frees the memory for the ContextInfo structure and any associated user memory as allocated with AllocContextInfo(). User purge vectors should call this function after they have freed any other resources associated with this structure.

Note that this function does NOT call the custom purge vector optionally installed through AllocContextInfo(); all it does is free the structure.

Arguments

ci
The ContextInfo structure to free. This may be NULL in which case this function does nothing.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

AllocContextInfo()

GetCurrentContext

Gets a pointer to the ContextNode for the current chunk.

Synopsis

```
ContextNode *GetCurrentContext(const IFFParser *iff);
```

Description

Returns the top ContextNode for the given parser handle. The top ContextNode corresponds to the chunk most recently pushed on the context stack, which is the chunk where the stream is currently positioned.

The ContextNode structure contains information on the type of chunk currently being processed, like its size and the current position within the chunk.

Arguments

iff

The parser handle to query.

Return Value

Returns a pointer to the current ContextNode for the parser.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

PushChunk(), PopChunk(), ParseIFF(), GetParentContext()

GetIFFOffset

Returns the absolute seek position within the current IFF stream.

Synopsis

```
int64 GetIFFOffset(IFFParser *iff);
```

Description

This function returns the count of bytes from the beginning of the current IFF stream. This is useful to identify the exact position of a chunk or part of a chunk within an IFF file.

Arguments

`iff`
The parsing handle to query.

Return Value

Returns the absolute seek offset from the beginning of the IFF stream to the current file cursor position, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

GetParentContext

Gets the nesting ContextNode for the given ContextNode.

Synopsis

```
ContextNode *GetParentContext (ContextNode *cn);
```

Description

This function returns a ContextNode for the chunk containing the chunk associated with the supplied ContextNode. This function effectively moves down the context stack into previously pushed contexts.

Arguments

cn
The ContextNode to obtain the parent of.

Return Value

Returns the parent of the supplied context or NULL if the supplied context has no parent.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

GetCurrentContext(), FindPropContext()

InstallEntryHandler

Adds an entry handler to the parser.

Synopsis

```
Err InstallEntryHandler(IFFParser *iff, PackedID type, PackedID id,  
                        ContextInfoLocation pos, IFFCallback cb,  
                        const void *userData);
```

Description

This function installs an entry handler for a specific type of chunk into the context for the given parser handle. The type and id are the identifiers for the chunk to handle.

The handler will be called whenever the parser enters a chunk of the given type, so the IFF stream will be positioned to read the first data byte in the chunk.

The value your callback routine returns affects the parser in two ways:

IFF_CB_CONTINUE

Normal return. The parser will continue through the file.

<any other value>

ParseIFF() will stop parsing and return this value directly to the caller. Return 0 for a normal return, and negative values for errors.

Arguments

iff

The parser handle to affect.

type

The ID of the container for the chunks to handle (eg: AIFF).

id

The ID value for the chunks to handle.

pos

Where the handler should be installed within the context stack. Refer to the StoreContextInfo() for a description of how this argument is used.

cb

The routine to invoke whenever a chunk of the given type and id is encountered.

userData

A value that is passed straight through to your callback routine when it is invoked.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

`InstallExitHandler()`

InstallExitHandler

Adds an exit handler to the parser.

Synopsis

```
Err InstallExitHandler(IFFParser *iff, PackedID type, PackedID id,  
                      ContextInfoLocation pos, IFFCallback cb,  
                      const void *userData);
```

Description

This function installs an exit handler for a specific type of chunk into the context for the given parser handle. The type and id are the identifiers for the chunk to handle.

The handler will be called whenever the parser is about to leave a chunk of the given type. The position within the stream is not constant and must be determined by looking at the `cn_Offset` field of the current context.

The value your callback routine returns affects the parser in two ways:

IFF_CB_CONTINUE

Normal return. The parser will continue through the file.

<any other value>

`ParseIFF()` will stop parsing and return this value directly to the caller. Return 0 for a normal return, and negative values for errors.

Arguments

`iff`

The parser handle to affect.

`type`

The ID of the container for the chunks to handle (eg: AIFF).

`id`

The ID value for the chunks to handle.

`pos`

Where the handler should be installed within the context stack. Refer to the `StoreContextInfo()` for a description of how this argument is used.

`cb`

The routine to invoke whenever a chunk of the given type and id is about to be exited.

`userData`

A value that is passed straight through to your callback routine when it is invoked.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

`InstallEntryHandler()`

ParseIFF

Parses an IFF stream.

Synopsis

```
Err ParseIFF(IFFParser *iff, ParseIFFModes control);
```

Description

This function traverses a stream opened for read by pushing chunks onto the context stack and popping them off directed by the generic syntax of IFF files. As it pushes each new chunk, it searches the context stack for handlers to apply to chunks of that type. If it finds an entry handler it will invoke it just after entering the chunk. If it finds an exit handler it will invoke it just before leaving the chunk. Standard handlers include entry handlers for pre-declared property chunks and collection chunks and entry and exit handlers for stop chunks - that is, chunks which will cause the `ParseIFF()` function to return control to the client. Client programs can also provide their own custom handlers.

The control argument can have one of three values:

IFF_PARSE_SCAN

In this normal mode, `ParseIFF()` will only return control to the caller when either:

- 1) an error is encountered, or
- 2) a stop chunk is encountered, or
- 3) a user handler returns a value other than 1, or
- 3) the end of the logical file is reached, in which case `IFF_PARSE_EOF` is returned.

`ParseIFF()` will continue pushing and popping chunks until one of these conditions occurs. If `ParseIFF()` is called again after returning, it will continue to parse the file where it left off.

IFF_PARSE_STEP and IFF_PARSE_RAWSTEP

In these two modes, `ParseIFF()` will return control to the caller after every step in the parse, specifically, after each push of a context node and just before each pop. If returning just before a pop, `ParseIFF()` will return `IFF_PARSE_EOC`, which is not an error, per se, but is just an indication that the most recent context is ending. In `STEP` mode, `ParseIFF()` will invoke the handlers for chunks, if any, before returning. In `RAWSTEP` mode, `ParseIFF()` will not invoke any handlers and will return right away. In both cases the function can be called multiple times to step through the parsing of the IFF file.

Arguments

`iff`

The parsing handle to affect.

`control`

Instructs how to parse.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

PushChunk(), PopChunk(), InstallEntryHandler(), InstallExitHandler(),
RegisterPropChunks(), RegisterCollectionChunks(), RegisterStopChunks()

PopChunk

Pops the top context node off the context stack.

Synopsis

```
Err PopChunk(IFFParser *iff);
```

Description

This function pops the top context chunk and frees all associated ContextInfo structures. The function is normally only called when writing files and signals the end of a chunk.

Arguments

iff
The parsing handle to affect.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

PushChunk()

PushChunk

Pushes a new context node on the context stack.

Synopsis

```
Err PushChunk(IFFParser *iff, PackedID type, PackedID id,  
              uint32 size);
```

Description

This function pushes a new context node on the context stack by reading it from the stream if this is a read stream, or by creating it from the passed parameters if this is a write stream. Normally this function is only called in write mode, where the type and id codes specify the new chunk to create. If this is a leaf chunk (i.e. a local chunk inside a FORM or PROP chunk), then the type argument is ignored.

If the size is specified then the chunk writing functions will enforce this size. If the size is given as IFF_SIZE_UNKNOWN_32, the chunk will expand to accommodate whatever is written into it up to a maximum of (IFF_SIZE_RESERVED-1) bytes. If the size is given as IFF_SIZE_UNKNOWN_64, then you can write as much data as there are atoms in the universe.

Arguments

iff	The parsing handle to affect.
type	The ID of the container (eg: AIFF). This is ignored in read mode, and for leaf chunks.
id	The chunk id specifier. This is ignored in read mode.
size	The size of the chunk to create, or IFF_SIZE_UNKNOWN_32 or IFF_SIZE_UNKNOWN_64. This parameter is ignored in read mode.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

PopChunk(), WriteChunk(), WriteChunkCompressed()

ReadChunk

Reads bytes from the current chunk into a buffer.

Synopsis

```
int32 ReadChunk(IFFParser *iff, void *buffer, uint32 numBytes);
```

Description

This function reads data from the IFF stream into the buffer for the specified number of bytes. Reads are limited to the size of the current chunk and attempts to read past the end of the chunk will truncate.

Arguments

`iff`

The parsing handle to read from.

`buffer`

A pointer to where the read data should be put.

`numBytes`

The number of bytes of data to read.

Return Value

Returns the actual number of bytes read, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

`ParseIFF()`, `ReadChunkCompressed()`, `SeekChunk()`, `WriteChunk()`,
`WriteChunkCompressed()`

ReadChunkCompressed

Reads and decompresses bytes from the current chunk into a buffer.

Synopsis

```
int32 ReadChunkCompressed(IFFParser *iff, void *buffer,  
                          uint32 numBytes);
```

Description

This function reads data from the IFF stream into the buffer for the specified number of bytes. Reads are limited to the size of the current chunk and attempts to read past the end of the chunk will truncate. As the data is read, it is automatically decompressed using the compression folio.

Arguments

- iff**
The parsing handle to read from.
- buffer**
A pointer to where the read data should be put.
- numBytes**
The number of bytes of data to read.

Return Value

Returns the actual number of bytes read, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

ParseIFF(), ReadChunk(), SeekChunk(), WriteChunk(),
WriteChunkCompressed()

RegisterCollectionChunks

Defines chunks to be collected during the parse operation.

Synopsis

```
Err RegisterCollectionChunks(IFFParser *iff, const IFFTypeID  
typeids[]);
```

Description

You supply this function an array of IFFTypeID structures which define the chunks that should be stored as they are encountered during the parse operation. The array is terminated by a structure containing a Type field set to 0.

The function installs an entry handler for chunks with the given type and id so that the contents of those chunks will be stored as they are encountered. This is similar to RegisterPropChunks() except that more than one chunk of any given type can be stored in lists which can be returned by FindCollection(). The storage of these chunks still follows the property chunk scoping rules for IFF files so that at any given point, stored collection chunks will be valid in the current context.

Arguments

iff

Parser handle to operate on.

typeids

An array of IFFTypeID structures defining which chunks should be collected. The array is terminated by a structure whose Type field is set to 0.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

IFF_ERR_NOMEM

There was not enough memory to perform the operation.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

FindCollection(), RegisterPropChunks()

RegisterPropChunks

Defines property chunks to be stored during the parse operation.

Synopsis

```
Err RegisterPropChunks(IFFParser *iff, const IFFTypeID typeids[]);
```

Description

You supply this function an array of IFFTypeID structures which define the chunks that should be stored as they are encountered during the parse operation. The array is terminated by a structure containing a Type field set to 0.

The function installs an entry handler for chunks with the given type and ID so that the contents of those chunks will be stored as they are encountered. The storage of these chunks follows the property chunk scoping rules for IFF files so that at any given point, a stored property chunk returned by FindPropContext() will be the valid property for the current context.

Arguments

iff

Parser handle to operate on.

typeids

An array of IFFTypeID structures defining which chunks should be stored. The array is terminated by a structure whose Type field is set to 0.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

IFF_ERR_NOMEM

There was not enough memory to perform the operation.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

FindPropContext()

RegisterStopChunks

Defines chunks that cause the parser to stop and return to the caller.

Synopsis

```
Err RegisterStopChunks(IFFParser *iff, const IFFTypeID typeids[]);
```

Description

You supply this function an array of IFFTypeID structures which define the chunks that should cause the parser to stop and return control to the caller. The array is terminated by a structure containing a Type field set to 0.

The function installs an entry handler for chunks with the given type and ID so that the parser will stop as they are encountered.

Arguments

iff

Parser handle to operate on.

typeids

An array of IFFTypeID structures defining which chunks should cause the parser to stop. The array is terminated by a structure whose Type field is set to 0.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible failure codes include:

IFF_ERR_NOMEM

There was not enough memory to perform the operation.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

ParseIFF()

RemoveContextInfo

Removes a ContextInfo structure from wherever it is attached.

Synopsis

```
void RemoveContextInfo(ContextInfo *ci);
```

Description

This function removes the ContextInfo structure from any list it might currently be in. If the structure is not currently in a list, this function is a NOP.

Arguments

`ci`
The ContextInfo structure to remove.

Implementation

Folio call implemented in IFF folio V28.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

AttachContextInfo(), StoreContextInfo(), AllocContextInfo(),
FreeContextInfo(), FindContextInfo()

SeekChunk

Moves the current position cursor within the current chunk.

Synopsis

```
Err SeekChunk(IFFParser *iff, int32 position, IFFSeekModes mode);
```

Description

This function moves the position cursor within the current chunk. The cursor determines where the next read operation will get its data from, and where the next write operation will write its data to.

The cursor can be moved to a position which is relative to the start or end of the current chunk, as well as relative to the current position. The cursor is never allowed to be less than 0 or greater than the number of bytes currently in the chunk. Any attempt to do so results in the cursor being positioned to the beginning or end of the chunk, depending on which limit was exceeded. This does not result in an error.

Arguments

iff

The parsing handle to seek in.

position

The position to move the cursor to.

mode

Describes what the position argument is relative to.

The possible seek modes are:

IFF_SEEK_START

Indicates that the supplied position is relative to the beginning of the chunk. So a position of 10 would put the cursor at byte #10 (the eleventh byte) within the chunk.

IFF_SEEK_CURRENT

Indicates that the supplied position is relative to the current position within the chunk. A positive position value moves the cursor forward in the file by that many bytes, while a negative value moves the cursor back by that number of bytes.

IFF_SEEK_END

Indicates that the supplied position is relative to the end of the chunk. The position value should be a negative value.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

ReadChunk(), ReadChunkCompressed(), WriteChunk(),
WriteChunkCompressed()

StoreContextInfo

Inserts a ContextInfo structure into the context stack.

Synopsis

```
Err StoreContextInfo(IFFParser *iff, ContextInfo *ci,  
                    ContextInfoLocation pos);
```

Description

This function adds the ContextInfo structure to the list of structures for one of the ContextNodes on the context stack and purges any other structure in the same context with the same type, id and ident codes. The pos argument determines where in the stack to add the structure:

IFF_CIL_BOTTOM

Add the structure to the ContextNode at the bottom of the stack.

IFF_CIL_TOP

Add the structure to the top (current) context node.

IFF_CIL_PROP

Add the structure in the top property context. The top property context is either the top FORM chunk, or the top LIST chunk, whichever is closer to the top of the stack.

Arguments**iff**

The parsing stream to affect.

ci

The ContextInfo structure to add.

pos

The position where the structure should be added.

Return ValueReturns ≥ 0 for success, or a negative error code for failure.**Implementation**

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

```
AllocContextInfo(),      FreeContextInfo(),      FindContextInfo(),  
AttachContextInfo(), RemoveContextInfo()
```

WriteChunk

Writes data from a buffer into the current chunk.

Synopsis

```
int32 WriteChunk(IFFParser *iff, const void *buffer,
                 uint32 numBytes);
```

Description

This function writes the requested number of bytes into the IFF stream. If the current chunk was pushed with IFF_SIZE_UNKNOWN_32 or IFF_SIZE_UNKNOWN_64, the size of the chunk gets increased by the size of the buffer written. If the size was specified for this chunk, attempts to write past the end of the chunk will be truncated.

Arguments

iff
The parsing handle to write to.

buffer
A pointer to the data to write to the stream.

numBytes
The number of bytes of data to write.

Return Value

Returns the number of bytes written, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

PushChunk(), PopChunk(), ReadChunk(), ReadChunkCompressed(), SeekChunk(),
WriteChunkCompressed()

WriteChunkCompressed

Compressed data from a buffer and writes the result to the current chunk.

Synopsis

```
int32 WriteChunkCompressed(IFFParser *iff, const void *buffer,  
                           uint32 numBytes);
```

Description

This function writes the requested number of bytes into the IFF stream. If the current chunk was pushed with IFF_SIZE_UNKNOWN_32 or IFF_SIZE_UNKNOWN_64, the size of the chunk gets increased by the size of the buffer written. If the size was specified for this chunk, attempts to write past the end of the chunk will be truncated. The data is first compressed using the compression folio before being written to the chunk.

Arguments

iff
The parsing handle to write to.

buffer
A pointer to the data to write to the stream.

numBytes
The number of bytes of data to write.

Return Value

Returns the number of bytes written, or a negative error code for failure.

Implementation

Folio call implemented in IFF folio V27.

Associated Files

<:misc:iff.h>, System.m2/Modules/iff

See Also

PushChunk(), PopChunk(), ReadChunk(), ReadChunkCompressed(), SeekChunk(), WriteChunk()

Chapter 12

International Folio Calls

This section presents the reference documentation for the International folio and associated link libraries.

intlCloseLocale

Terminates use of a given Locale item.

Synopsis

```
Err intlCloseLocale(Item locItem);
```

Description

This function concludes a client's use of the given Locale item. After this call is made, the Locale item may no longer be used.

Arguments

locItem
The Locale item, as obtained from `intlOpenLocale()`.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

INTL_ERR_BADITEM
locItem was not an existing Locale item.

INTL_ERR_CANTCLOSEITEM
An attempt was made to close this Locale item more often than it was opened.

Implementation

Macro implemented in `<:international:intl.h>` V24.

Associated Files

`<:international:intl.h>`, `System.m2/Modules/international`

See Also

`intlOpenLocale()`, `intlLookupLocale()`

intlCompareStrings

Compares two strings for collation purposes.

Synopsis

```
int32 intlCompareStrings(Item locItem, const unichar *string1,  
                          const unichar *string2);
```

Description

Compares two strings according to the collation rules of the Locale item's language.

Arguments

locItem
A Locale item, as obtained from `intlOpenLocale()`.

string1
The first string to compare.

string2
The second string to compare.

Return Value

-1
(string1 < string2)

0
(string1 == string2)

1
(string1 > string2)

INTL_ERR_BADITEM
locItem was not an existing Locale item.

Implementation

Folio call implemented in International folio V24.

Associated Files

<:international:intl.h>, System.m2/Modules/international

See Also

`intlOpenLocale()`, `intlConvertString()`

intlConvertString

Changes certain attributes of a string.

Synopsis

```
int32 intlConvertString(Item locItem, const unichar *string,  
                        unichar *result, uint32 resultSize,  
                        uint32 flags);
```

Description

Converts character attributes within a string. The flags argument specifies the type of conversion to perform.

Arguments

locItem

A Locale item, as obtained from intlOpenLocale().

string

The string to convert.

result

Where the result of the conversion is put. This area must be at least as large as the number of bytes in the string.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

flags

Description of the conversion process to apply:

INTL_CONVF_UPPERCASE will convert all characters to uppercase if possible.

INTL_CONVF_LOWERCASE will convert all characters to lowercase if possible.

INTL_CONVF_STRIPDIACRITICALS will remove diacritical marks from all characters.

INTL_CONVF_FULLWIDTH will convert all HalfKana characters to FullKana.

INTL_CONVF_HALFWIDTH will convert all FullKana characters to HalfKana.

You can also specify (INTL_CONVF_UPPERCASE|INTL_CONVF_STRIPDIACRITICALS) or (INTL_CONVF_LOWERCASE|INTL_CONVF_STRIPDIACRITICALS) in order to perform two conversions in one call.

If flags is 0, then a straight copy occurs.

Return Value

If positive, returns the number of characters in the result buffer. If negative, returns an error code. The string copied into the result buffer is guaranteed to be NULL-terminated. Possible error codes currently include:

INTL_ERR_BADSOURCEBUFFER

The string pointer supplied was bad.

INTL_ERR_BADRESULTBUFFER

The result buffer pointer was NULL or wasn't valid writable memory.

INTL_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

INTL_ERR_BADITEM

locItem was not an existing Locale item.

Implementation

Folio call implemented in International folio V24.

Associated Files

<:international:intl.h>, System.m2/Modules/international

Caveats

This function varies in intelligence depending on the language bound to the Locale argument. Specifically, most of the time, all characters above 0x0ff are not affected by the routine. The exception is with the Japanese language, where the Japanese characters are also affected by this routine.

See Also

intlOpenLocale(), intlCompareStrings()

intlFormatDate

Formats a date in a localized manner.

Synopsis

```
int32 intlFormatDate(Item locItem, DateSpec spec,  
                    const GregorianCalendar *date,  
                    unichar *result, uint32 resultSize);
```

Description

Formats a date according to a template and to the rules specified by the Locale item provided. The formatting string works similarly to `printf()` formatting strings, but uses specialized format commands tailored to date generation. The following format commands are supported:

%D - day of month

%H - hour using 24-hour style

%h - hour using 12-hour style

%M - minutes

%N - month name

%n - abbreviated month name

%O - month number

%P - AM or PM strings

%S - seconds

%W - weekday name

%w - abbreviated weekday name

%Y - year

In addition, the formatting string can also specify a field width, a field limit, and a field pad character, in a manner identical to the way `printf()` formatting strings specify these values:

%[flags][width][.limit]command

Flags can be "-" or "0", width is a positive numeric value, limit is a positive numeric value, and command is one of the format commands mentioned above. Refer to documentation on the standard C `printf()` function for more information on how these numbers and flags interact.

A difference with standard `printf()` processing is that the limit value is applied starting from the rightmost digits, instead of the leftmost. For example:

%0.2Y

Prints the rightmost two digits of the current year.

Arguments

`locItem`

A Locale item, as obtained from `intlOpenLocale()`.

`spec`

The formatting template describing the date layout. This value is typically taken from the Locale structure (`loc_Date`, `loc_ShortDate`, `loc_Time`, `loc_ShortTime`), but it can also be built up by hand for custom formatting.

`date`

The date to convert into a string.

`result`

Where the result of the formatting is put.

`resultSize`

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. The string copied into the result buffer guaranteed to be NULL-terminated. Possible error codes currently include:

`INTL_ERR_BADRESULTBUFFER`

The result buffer pointer was NULL or wasn't valid writable memory.

`INTL_ERR_BUFFERTOOSMALL`

There was not enough room in the result buffer.

`INTL_ERR_BADDATESPEC`

The pointer to the DateSpec array was bad.

`INTL_ERR_BADITEM`

`locItem` was not an existing Locale item.

`INTL_ERR_BADDATE`

The date specified in the `GregorianCalendar` structure is not a valid date. For example, the `gd_Month` is greater than 12.

Implementation

Folio call implemented in International folio V24.

Associated Files

<:international:intl.h>, System.m2/Modules/international

See Also

`intlOpenLocale()`, `intlFormatNumber()`

intlFormatNumber

Format a number in a localized manner.

Synopsis

```
int32 intlFormatNumber(Item locItem, const NumericSpec *spec,
                        uint32 whole, uint32 frac,
                        bool negative, bool doFrac,
                        unichar *result, uint32 resultSize);
```

Description

This function formats a number according to the rules contained in the `NumericSpec` structure. The `NumericSpec` structure is normally taken from a `Locale` structure. The `Locale` structure contains three initialized `NumericSpec` structures (`loc_Numbers`, `loc_Currency`, and `loc_SmallCurrency`) which let you format numbers in an appropriate manner for the system's current country selection.

You can create your own `NumericSpec` structure, which lets you use `intlFormatNumber()` to handle custom formatting needs. The fields in the structure have the following meaning:

`ns_PosGroups`

A `GroupingDesc` value defining how digits are grouped to the left of the decimal character. A `GroupingDesc` is simply a 32-bit bitfield. Every ON bit in the bitfield indicates that the separator sequence should be inserted after the associated digit. So if the third bit (bit #2) is ON, it means that the grouping separator should be inserted after the third digit of the formatted number.

`ns_PosGroupSep`

A string used to separate groups of digits to the left of the decimal character.

`ns_PosRadix`

A string used as a decimal character.

`ns_PosFractionalGroups`

A `GroupingDesc` value defining how digits are grouped to the right of the decimal character.

`ns_PosFractionalGroupSep`

A string used to separate groups of digits to the right of the decimal character.

`ns_PosFormat`

This field is used to do post-processing on a formatted number. This is typically used to add currency notation around a numeric value. The string in this field is used as a format string in a `sprintf()` function call, and the formatted numeric value is supplied as a parameter to the same `sprintf()` call. For example, if the `ns_PosFormat` field is defined as `"$%s"`, and the formatted numeric value is `"0.02"`, then the result of the post-processing will be `"$0.02"`. When this field is `NULL`, no post-processing occurs.

`ns_PosMinFractionalDigits`

Specifies the minimum number of digits to display to the right of the decimal character. If there are not enough digits, then the string will be padded on the right with 0s.

`ns_PosMaxFractionalDigits`

Specifies the maximum number of digits to display to the right of the decimal character. Any excess digits are just removed.

ns_NegGroups, ns_NegGroupSep, ns_NegRadix, ns_NegFractionalGroups,
ns_NegFractionalGroupSep, ns_NegFormat, ns_NegMinFractionalDigits,
ns_NegMaxFractionalDigits

These fields have the same function as the eight fields described above, except that they are used to process negative amounts, while the previous fields are used for positive amounts.

ns_Zero

If the number being processed is 0, then this string pointer is used as-is and is copied directly into the result buffer. If this field is NULL, the number is formatted as if it were a positive number.

ns_Flags

This is reserved for future use and must always be 0.

Arguments

locItem

A Locale Item, as obtained from `intlOpenLocale()`.

spec

The formatting template describing the number layout. This structure is typically taken from a Locale structure (`loc_Numbers`, `loc_Currency`, `loc_SmallCurrency`), but it can also be built up by hand for custom formatting.

whole

The whole component of the number to format. (The part of the number to the left of the radix character.)

frac

The decimal component of the number to format. (The part of the number to the right of the radix character.) This is specified in number of billionth. For example, to represent .5, you would use 500000000. To represent .0004, you would use 400000.

negative

TRUE if the number is negative, and FALSE if the number is positive.

doFrac

TRUE if a complete number with a decimal mark and decimal digits is desired. FALSE if only the whole part of the number should be output.

result

Where the result of the formatting is put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes which are put into the buffer.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. The string copied into the result buffer is guaranteed to be NULL-terminated. Possible error codes currently include:

INTL_ERR_BADNUMERICSPEC

The pointer to the NumericSpec structure was bad.

INTL_ERR_BADRESULTBUFFER

The result buffer pointer was NULL or wasn't valid writable memory.

INTL_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

INTL_ERR_BADITEM

locItem was not an existing Locale Item.

Implementation

Folio call implemented in International folio V24.

Associated Files

<:international:intl.h>, System.m2/Modules/international

See Also

intlOpenLocale(), intlFormatDate()

intlGetCharAttrs

Returns attributes describing a given character.

Synopsis

```
int32 intlGetCharAttrs(Item locItem, unichar character);
```

Description

This function examines the provided UniCode character and returns general information about it.

Arguments

locItem

A Locale item, as obtained from `intlOpenLocale()`.

character

The character to get the attribute of.

Return Value

Returns a bit mask, with bit sets to indicate various characteristics as defined by the UniCode standard. The possible bits are:

INTL_ATTRF_UPPERCASE

This character is uppercase.

INTL_ATTRF_LOWERCASE

This character is lowercase.

INTL_ATTRF_PUNCTUATION

This character is a punctuation mark.

INTL_ATTRF_DECIMAL_DIGIT

This character is a numeric digit.

INTL_ATTRF_NUMBER

This character represent a numerical value not representable as a single decimal digit. For example, the character 0x00bc represents the constant 1/2.

INTL_ATTRF_NONSPACING

This character is a nonspacing mark.

INTL_ATTRF_SPACE

This character is a space character.

INTL_ATTRF_HALF_WIDTH

This character is HalfKana.

INTL_ATTRF_FULL_WIDTH

This character is FullKana.

INTL_ATTRF_KANA

This character is Kana (Katakana).

INTL_ATTRF_HIRAGANA

This character is Hiragana.

INTL_ATTRF_KANJI

This character is Kanji.

If the value returned by this function is negative, then it is not a bit mask, and is instead an error code.

INTL_ERR_BADITEM

locItem was not an existing Locale item.

Implementation

Folio call implemented in International folio V24.

Associated Files

<:international:intl.h>, System.m2/Modules/international

Caveats

This function currently does not report any attributes for many upper UniCode characters. Only the ECMA Latin-1 character page (0x0000 to 0x00ff) is handled correctly at this time. If the language bound to the Locale structure is Japanese, then this function will also work correctly for Japanese characters.

See Also

intlOpenLocale(), intlConvertString()

intlLookupLocale

Returns a pointer to a Locale structure.

Synopsis

```
Locale *intlLookupLocale(Item locItem);
```

Description

This macro returns a pointer to a Locale structure. The structure can then be examined and its contents used to localize titles.

Arguments

locItem
A Locale item, as obtained from intlOpenLocale().

Return Value

The macro returns a pointer to a locale structure, which contains localization information, or NULL if the supplied Item was not a valid Locale item.

Implementation

Macro implemented in `<:international:intl.h>` V24.

Associated Files

`<:international:intl.h>`, `System.m2/Modules/international`

Example

```
{
Item      |t;
Locale *loc;

    it = intlOpenLocale(NULL);
    {
        loc = intlLookupLocale(it);

        // you can now read fields in the Locale structure.
        printf("Language is %ldn",loc->loc_Language);

        intlCloseLocale(it);
    }
}
```

See Also

`intlOpenLocale()`, `intlCloseLocale()`

intlOpenLocale

Gains access to a Locale item.

Synopsis

```
Item intlOpenLocale(const TagArg *tags);
```

Description

This function returns a Locale item. You can then use `intlLookupLocale()` to gain access to the localization information within the Locale item. This information enables software to adapt to different languages and customs automatically at run-time, thus creating truly international software.

Arguments

tags

A pointer to an array of tag arguments containing extra data for this function. This must currently always be NULL.

Return Value

The function returns the item number of a Locale. You can use `intlLookupLocale()` to get a pointer to the locale structure, which contains localization information.

Implementation

Macro implemented in `<:international:intl.h>` V24.

Associated Files

`<:international:intl.h>`, `System.m2/Modules/international`

Notes

Once you are finished with the Locale item, you should call `intlCloseLocale()`.

See Also

`intlCloseLocale()`, `intlLookupLocale()`

intlTransliterateString

Converts a string between character sets.

Synopsis

```
int32 intlTransliterateString(const void *string, CharacterSets
stringSet,
                                void *result, CharacterSets resultSet,
                                uint32 resultSize, uint8
unknownFiller);
```

Description

Converts a string between two character sets. This is typically used to convert a string from or to UniCode. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the destination character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
The string to transliterate.

stringSet
The character set of the string to transliterate. This describes the interpretation of the bytes in the source string.

result
A memory buffer where the result of the transliteration can be put.

resultSet
The character set to use for the resulting string.

resultSize
The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
If a character sequence cannot be adequately converted from the source character set, then this byte will be put into the result buffer in place of the character sequence. When converting to a 16-bit character set, then this byte will be extended to 16-bits and inserted.

Return Value

If positive, returns the number of characters in the result buffer. If negative, returns an error code. Possible error codes currently include:

INTL_ERR_BADSOURCEBUFFER
The string pointer supplied was bad.

INTL_ERR_BADCHARACTERSET
The "stringSet" or "resultSet" arguments did not specify valid character sets.

INTL_ERR_BADRESULTBUFFER
The result buffer pointer was NULL or wasn't valid writable memory.

INTL_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

INTL_ERR_BADITEM

locItem was not an existing Locale item.

Implementation

Folio call implemented in International folio V24.

Caveats

This function is not as smart as it could be. When converting from UniCode to ASCII, characters not in the first UniCode page (codes greater than 0x00ff) are never converted and always replaced with the unknownFiller byte. The upper pages of the UniCode set contain many characters which could be converted to equivalent ASCII characters, but these are not supported at this time.

Associated Files

<:international:intl.h>, System.m2/Modules/international

Chapter 13

JString Folio Calls

This section presents the reference documentation for the JString folio and associated link libraries.

ConvertASCII2ShiftJIS

Converts an ASCII string to Shift-JIS.

Synopsis

```
int32 ConvertASCII2ShiftJIS(const char *string,
                           char *result, uint32 resultSize,
                           uint8 unknownFiller);
```

Description

Converts a string from ASCII to Shift-JIS. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, returns the number of characters in the result buffer. If negative, returns an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertFullKana2HalfKana

Convert a full-width kana string to a half-width kana string.

Synopsis

```
int32 ConvertFullKana2HalfKana(const char *string,  
                               char *result, uint32 resultSize,  
                               uint8 unknownFiller);
```

Description

Converts a string from full-width kana to half-width kana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertFullKana2Hiragana

Convert a full-width kana string to a hiragana string.

Synopsis

```
int32 ConvertFullKana2Hiragana(const char *string,  
                               char *result, uint32 resultSize,  
                               uint8 unknownFiller);
```

Description

Converts a string from full-width kana to hiragana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

- string**
The string to convert.
- result**
A memory buffer where the result of the conversion can be put.
- resultSize**
The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.
- unknownFiller**
If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL
There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertFullKana2Romaji

Convert a full-width kana string to a romaji string.

Synopsis

```
int32 ConvertFullKana2Romaji(const char *string,  
                             char *result, uint32 resultSize,  
                             uint8 unknownFiller);
```

Description

Converts a string from full-width kana to romaji. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertHalfKana2FullKana

Convert a half-width kana string to a full-width kana string.

Synopsis

```
int32 ConvertHalfKana2FullKana(const char *string,  
                               char *result, uint32 resultSize,  
                               uint8 unknownFiller);
```

Description

Converts a string from half-width kana to full-width kana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertHalfKana2Hiragana

Convert a half-width kana string to a hiragana string.

Synopsis

```
int32 ConvertHalfKana2Hiragana(const char *string,  
                               char *result, uint32 resultSize,  
                               uint8 unknownFiller);
```

Description

Converts a string from half-width kana to hiragana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertHalfKana2Romaji

Convert a half-width kana string to a romaji string.

Synopsis

```
int32 ConvertHalfKana2Romaji(const char *string,  
                             char *result, uint32 resultSize,  
                             uint8 unknownFiller);
```

Description

Converts a string from half-width kana to romaji. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertHiragana2FullKana

Convert a hiragana string to a full-width kana string.

Synopsis

```
int32 ConvertHiragana2FullKana(const char *string,  
                               char *result, uint32 resultSize,  
                               uint8 unknownFiller);
```

Description

Converts a string from hiragana to full-width kana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertHiragana2HalfKana

Convert a hiragana string to a half-width kana string.

Synopsis

```
int32 ConvertHiragana2HalfKana(const char *string,  
                               char *result, uint32 resultSize,  
                               uint8 unknownFiller);
```

Description

Converts a string from hiragana to half-width kana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

- string**
The string to convert.
- result**
A memory buffer where the result of the conversion can be put.
- resultSize**
The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.
- unknownFiller**
If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL
There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertHiragana2Romaji

Convert a hiragana string to a romaji string.

Synopsis

```
int32 ConvertHiragana2Romaji(const char *string,
                             char *result, uint32 resultSize,
                             uint8 unknownFiller);
```

Description

Converts a string from hiragana to romaji. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments**string**

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertRomaji2FullKana

Convert a romaji string to a full-width kana string.

Synopsis

```
int32 ConvertRomaji2FullKana(const char *string,  
                             char *result, uint32 resultSize,  
                             uint8 unknownFiller);
```

Description

Converts a string from romaji to full-width kana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string
The string to convert.

result
A memory buffer where the result of the conversion can be put.

resultSize
The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller
If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL
There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertRomaji2HalfKana

Convert a romaji string to a half-width kana string.

Synopsis

```
int32 ConvertRomaji2HalfKana(const char *string,
                             char *result, uint32 resultSize,
                             uint8 unknownFiller);
```

Description

Converts a string from romaji to half-width kana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

- string**
The string to convert.
- result**
A memory buffer where the result of the conversion can be put.
- resultSize**
The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.
- unknownFiller**
If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL
There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<.:international:jstring.h>, System.m2/Modules/jstring

ConvertRomaji2Hiragana

Convert a romaji string to a hiragana string.

Synopsis

```
int32 ConvertRomaji2Hiragana(const char *string,  
                             char *result, uint32 resultSize,  
                             uint8 unknownFiller);
```

Description

Converts a string from romaji to hiragana. The conversion is done as well as possible. If certain characters from the source string cannot be represented in the target character set, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertShiftJIS2ASCII

Converts a Shift-JIS string to ASCII.

Synopsis

```
int32 ConvertShiftJIS2ASCII(const char *string,  
                           char *result, uint32 resultSize,  
                           uint8 unknownFiller);
```

Description

Converts a string from Shift-JIS to ASCII. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

- string**
The string to convert.
- result**
A memory buffer where the result of the conversion can be put.
- resultSize**
The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.
- unknownFiller**
If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL
There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<.:international:jstring.h>, System.m2/Modules/jstring

ConvertShiftJIS2Unicode

Converts a Shift-JIS string to UniCode.

Synopsis

```
int32 ConvertShiftJIS2Unicode(const char *string,  
                             unichar *result, uint32 resultSize,  
                             uint8 unknownFiller);
```

Description

Converts a string from Shift-JIS to UniCode. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments**string**

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

ConvertUnicode2ShiftJIS

Converts a UniCode string to Shift-JIS.

Synopsis

```
int32 ConvertUnicode2ShiftJIS(const unichar *string,  
                             char *result, uint32 resultSize,  
                             uint8 unknownFiller);
```

Description

Converts a string from UniCode to Shift-JIS. The conversion is done as well as possible. If certain characters from the source string cannot be represented in ASCII, the unknownFiller byte will be inserted in the result buffer in their place.

Arguments

string

The string to convert.

result

A memory buffer where the result of the conversion can be put.

resultSize

The number of bytes available in the result buffer. This limits the number of bytes that are put into the buffer.

unknownFiller

If a character sequence cannot be adequately converted, then this byte will be put into the result buffer in place of the character sequence.

Return Value

If positive, then the number of characters in the result buffer. If negative, then an error code. Possible error codes currently include:

JSTR_ERR_BUFFERTOOSMALL

There was not enough room in the result buffer.

Implementation

Folio call implemented in Jstring folio V24.

Associated Files

<:international:jstring.h>, System.m2/Modules/jstring

Chapter 14

Kernel Folio Calls

This section presents the reference documentation for the Kernel folio and associated link libraries.

AtomicClearBits

Clears bit in a word of memory in an atomic manner.

Synopsis

```
uint32 AtomicClearBits(uint32 *addr, uint32 bits);
```

Description

This function clears the specified bits from the given word of memory in an atomic manner. Essentially, you are guaranteed that the write operation will be performed without interference from other tasks. This is a very efficient replacement for many common uses of semaphores.

Arguments

addr

The address of the word of memory to modify.

bits

The bits in the word of memory that should be cleared.

Return Value

Returns the previous contents of the word of memory.

Implementation

Link library call implemented in libc.a V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

AtomicSetBits()

AtomicSetBits

Sets bit in a word of memory in an atomic manner.

Synopsis

```
uint32 AtomicSetBits(uint32 *addr, uint32 bits);
```

Description

This function sets the specified bits in the given word of memory in an atomic manner. Essentially, you are guaranteed that the write operation will be performed without interference from other tasks. This is a very efficient replacement for many common uses of semaphores.

Arguments

addr
The address of the word of memory to modify.

bits
The bits in the word of memory that should be set.

Return Value

Returns the previous contents of the word of memory.

Implementation

Link library call implemented in libc.a V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

AtomicClearBits()

ClearBitRange

Clear a range of bits within a bit array.

Synopsis

```
void ClearBitRange(uint32 *array, uint32 firstBit, uint32 lastBit)
```

Desription

This function turns a range of bits off within a bit array. All bits within the range [firstBit..lastBit] are set to 0.

firstBit must be <= to lastBit, otherwise the results are undefined.

Arguments

array The array of bits.

firstBit The first bit within the range to clear.

lastBit The last bit within the range to clear.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), IsBitRangeSet(), IsBitRangeClear(), IsBitSet(),
IsBitClear(), FindSetBitRange(), FindClearBitRange(), DumpBitRange()

CountBits

Count the number of bits set in a word.

Synopsis

```
uint32 CountBits(uint32 mask);
```

Description

This function counts the number of bits set in the supplied 32-bit word using a clever algorithm.

Arguments

mask
The word to count the bits of.

Return Value

The number of bits that were set in the supplied value.

Implementation

Link library call implemented in libc.a V21.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

FindMSB(), FindLSB()

DumpBitRange

Display a range of bits to the debugging terminal.

Synopsis

```
void DumpBitRange(const uint32 *array, uint32 firstBit,  
                  uint32 lastBit, const char *banner)
```

Description

This function sends a range of bits within a bit array to the debugging terminal. This is useful during debugging.

Arguments

- | | |
|----------|---|
| array | The array of bits. |
| firstBit | Marks the beginning of the range within the bit array to dump. |
| lastBit | Marks the end of the range within the bit array to dump. |
| banner | Descriptive text dumped before the main output, used to identify the origin of this call.
May be NULL. |

Implementation

Library call implemented in libc.a V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), ClearBitRange(), IsBitRangeSet(), IsBitRangeClear(),
IsBitSet(), IsBitClear(), FindSetBitRange(), FindClearBitRange()

FindClearBitRange

Find a range of clear bits within a bit array.

Synopsis

```
int32 FindClearBitRange(const uint32 *array, uint32 firstBit,  
                        uint32 lastBit, uint32 numBits)
```

Description

This function searches through a bit array for a range of bits that are all set to 0. The search is limited to bits within [firstBit..lastBit].

firstBit must be <= to lastBit, otherwise the results are undefined.

Arguments

array

The array of bits.

firstBit

Marks the beginning of the search range within the bit array.

lastBit

Marks the end of the search range within the bit array.

numBits

The number of bits in a row that must be clear. If this value is 0, this function always returns -1.

Return Value

This function returns the number of the first bit within the range of clear bits that was found, or -1 if there was no sufficiently large range of clear bits within the bit array.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), ClearBitRange(), IsBitRangeSet(), IsBitRangeClear(),
IsBitSet(), IsBitClear(), FindSetBitRange(), DumpBitRange()

FindLSB

Finds the least-significant bit.

Synopsis

```
int32 FindLSB(uint32 mask);
```

Description

`FindLSB()` finds the lowest-numbered bit that is set in the argument. The least-significant bit is bit 0, and the most-significant bit is bit 31.

Arguments

`mask`
The 32-bit word to check.

Return Value

Returns the number of the lowest-numbered bit that is set in the argument, or -1 if no bits are set.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

`<:kernel:bitarray.h>`, `libc.a`

See Also

`FindMSB()`

FindMSB

Finds the highest-numbered bit.

Synopsis

```
int32 FindMSB(uint32 mask);
```

Description

This function finds the highest-numbered bit that is set in the argument. The least-significant bit is bit 0, and the most-significant bit is bit 31.

Arguments

mask
The 32-bit word to check.

Return Value

Returns the number of the highest-numbered bit that is set in the argument, or -1 if no bits are set.

Implementation

Link library call implemented in libc.a V20.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

FindLSB()

FindSetBitRange

Find a range of set bits within a bit array.

Synopsis

```
int32 FindSetBitRange(const uint32 *array, uint32 firstBit,  
                     uint32 lastBit, uint32 numBits)
```

Description

This function searches through a bit array for a range of bits that are all set to 1. The search is limited to bits within [firstBit..lastBit].

firstBit must be <= to lastBit, otherwise the results are undefined.

Arguments

- array
The array of bits.
- firstBit
Marks the beginning of the search range within the bit array.
- lastBit
Marks the end of the search range within the bit array.
- numBits
The number of bits in a row that must be set. If this value is 0, this function always returns -1.

Return Value

This function returns the number of the first bit within the range of set bits that was found, or -1 if there was no sufficiently large range of set bits within the bit array.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), ClearBitRange(), IsBitRangeSet(), IsBitRangeClear(),
IsBitSet(), IsBitClear(), FindClearBitRange(), DumpBitRange()

IsBitClear

Test whether a bit within a bit array is set to 0.

Synopsis

```
bool IsBitClear(const uint32 *array, uint32 bit)
```

Description

This function lets you test to see if a single bit within a bit array is set to 0.

Arguments

array
The array of bits.

bit
The bit number to test.

Return Value

This function returns TRUE if the given bit is set to 0, and return FALSE if it is set to 1.

Implementation

Macro implemented in `<:kernel:bitarray.h>` V27.

Associated Files

`<:kernel:bitarray.h>`, `libc.a`

See Also

`SetBitRange()`, `ClearBitRange()`, `IsBitRangeSet()`, `IsBitRangeClear()`,
`IsBitSet()`, `FindSetBitRange()`, `FindClearBitRange()`, `DumpBitRange()`

IsBitRangeClear

Test whether all bits within a range of a bit array are all set to 0.

Synopsis

```
bool IsBitRangeClear(const uint32 *array, uint32 firstBit,  
                    uint32 lastBit)
```

Description

This function lets you test to see if all bits within a range of bits in a bit array are all set to 0.

firstBit must be <= to lastBit, otherwise the results are undefined.

Arguments

array
The array of bits.

firstBit
The first bit within the range to test.

lastBit
The last bit within the range to test.

Return Value

This function returns TRUE if all bits within the range [firstBit..lastBit] are set to 0, and FALSE otherwise.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), ClearBitRange(), IsBitRangeSet(), IsBitSet(),
IsBitClear(), FindSetBitRange(), FindClearBitRange(), DumpBitRange()

IsBitRangeSet

Test whether all bits within a range of a bit array are all set to 1.

Synopsis

```
bool IsBitRangeSet(const uint32 *array, uint32 firstBit,  
                  uint32 lastBit)
```

Description

This function lets you test to see if all bits within a range of bits in a bit array are all set to 1.

firstBit must be \leq to lastBit, otherwise the results are undefined.

Arguments

array

The array of bits.

firstBit

The first bit within the range to test.

lastBit

The last bit within the range to test.

Return Value

This function returns TRUE if all bits within the range [firstBit..lastBit] are set to 1, and FALSE otherwise.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), ClearBitRange(), IsBitRangeClear(), IsBitSet(),
IsBitClear(), FindSetBitRange(), FindClearBitRange(), DumpBitRange()

IsBitSet

Test whether a bit within a bit array is set to 1.

Synopsis

```
bool IsBitSet(const uint32 *array, uint32 bit)
```

Description

This function lets you test to see if a single bit within a bit array is set to 1.

Arguments

array
The array of bits.

bit
The bit number to test.

Return Value

This function returns TRUE if the given bit is set to 1, and return FALSE if it is set to 0.

Implementation

Macro implemented in *<:kernel:bitarray.h>* V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

SetBitRange(), ClearBitRange(), IsBitRangeSet(), IsBitRangeClear(),
IsBitClear(), FindSetBitRange(), FindClearBitRange(), DumpBitRange()

SetBitRange

Set a range of bits within a bit array.

Synopsis

```
void SetBitRange(uint32 *array, uint32 firstBit, uint32 lastBit)
```

Description

This function turns a range of bits on within a bit array. All bits within the range [firstBit..lastBit] are set to 1.

firstBit must be \leq to lastBit, otherwise the results are undefined.

Arguments

array

The array of bits.

firstBit

The first bit within the range to set.

lastBit

The last bit within the range to set.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:bitarray.h>, libc.a

See Also

ClearBitRange(), IsBitRangeSet(), IsBitRangeClear(), IsBitSet(),
IsBitClear(), FindSetBitRange(), FindClearBitRange(), DumpBitRange()

FlushDCache

Write back modified contents of the data cache to memory, and remove the data from the cache.

Synopsis

```
void FlushDCache(uint32 flushCount, const void *start,
                 uint32 numBytes);
```

Description

This function writes back data to main memory from the CPU data cache, and then removes the data from the cache. The writes only occur if the data has been modified and not written back to memory previously. Only the data in the supplied address range is affected.

The flush count argument comes from `GetDCacheFlushCount()`. If the flush count you supply to this function doesn't match the current system flush count, then this function is a NOP. This helps improve system performance.

In order to benefit from the flush count strategy, you should call `GetDCacheFlushCount()` as soon as the CPU is done manipulating the data area of interest, and you should call `FlushDCache()` at the last moment possible.

Arguments

`flushCount`

A flush count value obtained from `GetDCacheFlushCount()`.

`start`

The starting address in the memory range to make sure is written back and removed from the cache.

`numBytes`

The number of bytes in the memory range.

Implementation

Folio call implemented in Kernel folio V27.

Warning

This function should very seldom be used by application code. Abusing this function will cause a tremendous decrease in system performance.

Associated Files

`<:kernel:cache.h>`, `libc.a`

See Also

`GetDCacheFlushCount()`, `WriteBackDCache()`

FlushDCacheAll

Write back modified contents of the data cache to memory, and remove the data from the cache.

Synopsis

```
void FlushDCacheAll(uint32 flushCount);
```

Description

This function writes back data to main memory from the CPU data cache, and then removes the data from the cache. The writes only occur if the data has been modified and not written back to memory previously.

The flush count argument comes from `GetDCacheFlushCount()`. If the flush count you supply to this function doesn't match the current system flush count, then this function is a NOP. This helps improve system performance.

In order to benefit from the flush count strategy, you should call `GetDCacheFlushCount()` as soon as the CPU is done manipulating the data area of interest, and you should call `FlushDCacheAll()` at the last moment possible.

Arguments

`flushCount`

A flush count value obtained from `GetDCacheFlushCount()`.

Implementation

Folio call implemented in Kernel folio V27.

Warning

This function should very seldom be used by application code. Abusing this function will cause a tremendous decrease in system performance.

Associated Files

`<:kernel:cache.h>`, `libc.a`

See Also

`GetDCacheFlushCount()`, `WriteBackDCache()`, `FlushDCache()`

GetCacheInfo

Gets information about the CPU caches.

Synopsis

```
void GetCacheInfo(CacheInfo *info, uint32 infoSize);
```

Description

This function returns information about the CPU caches. The information includes their size and their state. The info is returned in a CacheInfo structure, containing the following fields:

`info_Flags`

Provides a number of flag that show the state of the caches. See the `CACHE_XXX` flags in `<:kernel:cache.h>`.

`info_ICacheSize` and `info_DCacheSize`

Specifies the number of bytes in the primary CPU instruction and data caches.

`info_ICacheLineSize` and `info_DCacheLineSize`

Specifies the number of bytes in one line of the primary CPU instruction and data cache.

`info_ICacheSetSize` and `info_DCacheSetSize`

Specifies the level of set associativity implemented in the primary CPU instruction and data cache.

Arguments

`info`

Pointer to a structure where the cache information will be put.

`infoSize`

Size of the structure to receive the cache information.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:kernel:cache.h>`, `libc.a`

See Also

`FlushDCache()`

GetDCacheFlushCount

Obtain the current system cache flush count.

Synopsis

```
uint32 GetDCacheFlushCount(void);
```

Description

This function returns the system's current data cache flush count. Everytime the data cache is completely flushed during the course of running the system, a count is incremented.

In order to flush the data cache, you must supply a flush count to the `FlushDCache()` and `WriteBackDCache()` functions. If the flush count you supply differs from the system's current flush count, then no flushing will occur.

The intent behind maintaining the flush count is to prevent needless cache flushes by leveraging off of other flushes that are occurring asynchronously elsewhere in the system.

For example, once some data has been prepared for a DMA operation, you get the current flush count. When time actually comes to hand the data to the DMA hardware, you attempt to flush the caches to make sure your data hits memory. If the flush count you supply differs from the current system flush count, it means that some other entity in the system has already flushed the caches, so your call to `FlushDCache()` or `WriteBackDCache()` will be ignored.

Return Value

Returns the current system data cache flush count.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:kernel:cache.h>`, `libc.a`

See Also

`WriteBackDCache()`, `FlushDCache()`

WriteBackDCache

Write back modified contents of the data cache to memory.

Synopsis

```
void WriteBackDCache(uint32 flushCount, const void *start,  
                     uint32 numBytes);
```

Description

This function writes back data to main memory from the CPU data cache. The writes only occur if the data has been modified and not written back to memory previously. Only the data in the supplied address range is affected.

The flush count argument comes from `GetDCacheFlushCount()`. If the flush count you supply to this function doesn't match the current system flush count, then this function is a NOP. This helps improve system performance.

In order to benefit from the flush count strategy, you should call `GetDCacheFlushCount()` as soon as the CPU is done manipulating the data area of interest, and you should call `WriteBackDCache()` at the last moment possible.

Arguments

`flushCount`

A flush count value obtained from `GetDCacheFlushCount()`.

`start`

The starting address in the memory range to make sure is written back.

`numBytes`

The number of bytes in the memory range.

Implementation

Folio call implemented in Kernel folio V27.

Warning

This function should very seldom be used by application code. Abusing this function will cause a tremendous decrease in system performance.

Associated Files

<:kernel:cache.h>, libc.a

See Also

`GetDCacheFlushCount()`, `FlushDCache()`

ScanForDDFToken

Scan for a particular token in a token sequence.

Synopsis

```
Err ScanForDDFToken(void* np, char const* name, DDFTokenSeq*
rhstokens);
```

Description

This is a super vector that may be called in user mode. The input token sequence is searched for a token whose name matches the specified parameter. If the token is found, the "right-hand-side" token sub-sequence whose first token is the specified token is returned in the DDFTokenSeq buffer. If the token is not found, the buffer is unchanged.

Arguments

np
Pointer to the token sequence to search.

name
The name of the token for which to search.

rhstokens
DDFTokenSeq token sequence buffer that will accept the sub-sequence.

Return Value

Returns zero or a negative error code for failure. Possible error codes currently include:

ER_NotFound
No token with the given name was encountered.

ER_SoftErr
An unknown token was encountered in the sequence.

Implementation

Folio call implemented in kernel folio V28.

Associated Files

<:kernel:super.h>

See Also

NextDDFToken()

ControlIODebug

Controls what IODebug does and doesn't do.

Synopsis

```
Err ControlIODebug(bool active);
```

Description

This function lets you control various options that determine what IODebug does.

IODebug can do the following things:

- When writing data to an I/O device, checks are done to ensure that the buffer used for the I/O remains consistent and does not get altered while the write operation occurs. This guarantees that a client is not fiddling with data that is in the process of being fetched by an I/O module.
- When reading data from an I/O device, prevents any data associated with the I/O request from being read until the I/O request has completed. This guarantees that a client is not reading data before it has been completely transferred.

IODebug does its work by allocating temporary buffers and using these for the I/O transactions. Doing so requires some extra memory, and requires some extra CPU time to copy data between the buffers. This can affect the performance of a program, possibly causing some jerky animations, but should never cause a crash, or trashed sound or graphics. If these things happen, then the program has a bug.

Arguments

controlFlags

A set of bit flags controlling various IODebug options. See below.

Flags

The control flags can be any of:

IODEBUGF_PREVENT_PREREAD

Makes sure that client tasks don't attempt to read data from an I/O buffer supplied using the IOInfo.ioi_Recv field while the I/O operation is still in progress. This catches bugs where data is fetched prematurely.

IODEBUGF_PREVENT_POSTWRITE

Makes sure that client tasks don't attempt to write data to an I/O buffer supplied using the IOInfo.ioi_Send field while the I/O operation is still in progress. This catches bugs where data is modified before it has been completely processed by an I/O device.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V30.

Associated Files

<:kernel:io.h>

DebugBreakpoint

Trigger a breakpoint-like event in the debugger.

Synopsis

```
void DebugBreakpoint(void);
```

Description

This function lets you tell the debugger to interrupt execution of your task just like if a breakpoint was in place. The debugger will then be in control, and normal debugging activities can be performed, just like when a breakpoint is encountered.

Implementation

Macro implemented in `<:kernel:debug.h>` V27.

Associated Files

`<:kernel:debug.h>`, `libc.a`

See Also

`DebugPutStr()`, `DebugPutChar()`

DebugPutChar

Output a character to the debugging terminal.

Synopsis

```
void DebugPutChar(char *ch);
```

Description

This function outputs a character to the debugging terminal. If there is no debugging terminal, this function has no effect.

Arguments

ch

The character to output to the debugging terminal.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:debug.h>, libc.a

See Also

DebugPutStr(), DebugBreakpoint()

DebugPutStr

Output a string to the debugging terminal.

Synopsis

```
void DebugPutStr(const char *str);
```

Description

This function outputs a string to the debugging terminal. If there is no debugging terminal, this function has no effect.

Arguments

str
The string to output.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:debug.h>, libc.a

See Also

DebugPutChar(), DebugBreakpoint()

GetSysErr

Gets the error string for an error.

Synopsis

```
int32 GetSysErr(char *buff, int32 bufsize, Err err);
```

Description

This function returns a character string that describes a Portfolio error code. The resulting string is placed in a buffer.

Arguments

buff

A pointer to a buffer to hold the error string.

bufsize

The size of the error-text buffer, in bytes. This should be at least 128.

err

The error code whose error string to get.

Return Value

Returns the number copied to the description buffer, or a negative error code if a bad buffer pointer is supplied.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:operror.h>, libc.a

See Also

PrintfSysErr()

PrintfSysErr

Prints the error string for an error.

Synopsis

```
void PrintfSysErr(Err err);
```

Description

Portfolio provides a uniform error management scheme. Errors are denoted by negative values, and each error code is unique in the system, allowing an easy determination of what caused the error to occur.

Each error code in Portfolio has a string associated with it that attempts to describe the error in human terms. This function takes a Portfolio error code and displays its string to the debugging terminal.

Arguments

err

An error code as obtained from many Portfolio system calls.

Implementation

Folio call implemented in Kernel folio V30.

Associated Files

<:kernel:operror.h>, libc.a

See Also

GetSysErr()

CloseDeviceStack

Closes a stack of devices.

Synopsis

```
Err CloseDeviceStack(Item devItem);
```

Description

Closes a stack of devices previously opened by OpenDeviceStack.

Arguments

devItem

Item number of the DeviceStack to be closed.

Return Value

Returns zero or a negative error code for failure. Possible error codes currently include:

TBD

Implementation

Folio call implemented in kernel folio V28.

Associated Files

<:kernel:device.h> *<:kernel:ddfnode.h>*

See Also

OpenDeviceStack(), CreateDeviceStackList()

CreateDeviceStackList

Get a list of device stacks which support the requested capabilities.

Synopsis

```
Err CreateDeviceStackList(List **pList, const DDFQuery query[]);
Err CreateDeviceStackListVA(List **pList, const char *name, DDFOp
op, uint32 flags, uint32 numValues, ...);
```

Description

The list of all devices currently supported by the system is searched for those devices which support all of the capabilities in the provided array. The returned list of DeviceStacks describes all stacks of devices which support the specified capabilities.

The query is an array of DDFQuery structures. In each, the `ddfq_Name` field is the name of a device capability; the `ddfq_Operator` field is a matching operator (DDFQ_EQ for equality, DDFQ_GT for "greater than", etc.); the `ddfq_Value` field is the value to be matched; and the `ddfq_Flags` field is set to DDF_INT if the `ddfq_Value` field is an integer (int32) or DDF_STRING if it is a string (char *).

CreateDeviceStackListVA() provides a simplified query interface. The capability to be queried is specified by the "name" argument. The "op" argument specifies the matching operator to be applied to each value. The numValues argument specifies the number of values following the flags argument. If the flags argument is DDF_INT, the values are taken to be int32 values; if it is DDF_STRING they are taken to be char * values. Following the last value argument, this entire sequence of arguments (name, op, flags, numValues, and a list of values) may be repeated to query additional capabilities. The last value argument for the last capability is followed by a (char*)NULL in place of the name argument.

Arguments (CreateDeviceStackList)

pList	Pointer to a List * which upon successful completion is set to point to a list of DeviceStacks. If no devices in the system satisfy the query, the List will be empty.
query	Array of device description capability requirements.

Arguments (CreateDeviceStackListVA)

pList	Same as in CreateDeviceStackList.
name	Name of device capability.
op	Operator to be applied to each value (one of DDF_EQ, DDF_GT, DDF_LT, DDF_GE, DDF_LE, DDF_NE).
flags	DDF_INT if values are int32s, DDF_STRING if values are char *'s.
numValues	Number of value arguments.

Return Value

Returns zero or a negative error code for failure. Possible error codes currently include:

TBD

Implementation

Folio call implemented in kernel folio V28.

Associated Files

`<:kernel:device.h>` `<:kernel:ddfnode.h>`

See Also

`OpenDeviceStack()`, `CloseDeviceStack()`, `DeleteDeviceStackList()`

DeleteDeviceStackList

Destroy the List created by
CreateDeviceStackList().

Synopsis

```
void DeleteDeviceStackList(List* l);
```

Description

Deletes a list previously returned from CreateDeviceStackList. The list itself, and all DeviceStacks in it, are deleted.

Arguments

l
Pointer to the List to be deleted.

Implementation

Folio call implemented in kernel folio V28.

Associated Files

<:kernel:device.h> <:kernel:ddfnode.h>

See Also

CreateDeviceStackList()

DeviceStackIsIdentical

Compare a DeviceStack to an open device stack.

Synopsis

```
bool DeviceStackIsIdentical(DeviceStack *ds, Item devItem);
```

Description

Compares the specified DeviceStack to the specified item, which should be an open device stack, returned from `OpenDeviceStack()`.

Arguments

`ds`
Pointer to the DeviceStack to be compared.

`devItem`
Item number of the open device stack to be compared.

Return Value

Returns TRUE if the two device stacks are identical (from top to bottom).

Implementation

Folio call implemented in kernel folio V28.

Associated Files

`<:kernel:device.h>` `<:kernel:ddfnode.h>`

See Also

`CreateDeviceStackList()`, `OpenDeviceStack()`

OpenDeviceStack

Opens a stack of devices.

Synopsis

```
Item OpenDeviceStack(DeviceStack *ds);
```

Description

Opens a stack of devices. This stack is normally one node from a list of DeviceStacks returned from CreateDeviceStackList.

Arguments

ds
Pointer to the DeviceStack to be opened.

Return Value

Returns the item number of the opened device stack, or an error code if there was an error in opening the device stack.

Implementation

Folio call implemented in kernel folio V28.

Associated Files

<:kernel:device.h> <:kernel:ddfnode.h>

See Also

CloseDeviceStack(), CreateDeviceStackList()

AbortIO

Aborts an I/O operation.

Synopsis

```
Err AbortIO(Item ior);
```

Description

This function aborts an I/O operation. If the I/O operation has already completed, calling `AbortIO()` has no effect. If it is not complete, it will be aborted.

A task should call `WaitIO()` immediately after calling `AbortIO()`. When `WaitIO()` returns, the task knows that the I/O operation is no longer active. It can then confirm that the I/O operation was aborted before it was finished by checking the `io_Error` field of the `IOReq` structure. If the operation aborted, the value of this field is `ER_Aborted`.

Arguments

`ior`

The item number of the I/O request to abort.

Return Value

Returns 0 if the I/O operation was successfully aborted or a negative error code for failure. Possible error codes currently include:

`BADITEM`

The `ior` argument is not an `IOReq`.

`NOTOWNER`

The `ior` argument is an `IOReq` but the calling task is not its owner.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:io.h>`, `libc.a`

See Also

`CheckIO()`, `CreateIOReq()`, `DeleteIOReq()`, `DoIO()`, `SendIO()`, `WaitIO()`

CheckIO

Checks whether an I/O operation is complete.

Synopsis

```
int32 CheckIO(Item ior);
```

Description

This function checks to see if an I/O operation has completed.

Arguments

ior
The item number of the I/O request to be checked.

Return Value

Returns 0 if the I/O is not complete. It returns > 0 if it is complete. It returns BADITEM if ior is bad.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

<:kernel:io.h>, libc.a

See Also

AbortIO(), CreateIOReq(), DeleteIOReq(), DoIO(), SendIO(), WaitIO()

CreateIOReq

Creates an I/O request.

Synopsis

```
Item CreateIOReq(const char *name, uint8 pri, Item dev,  
                Item msgPort);
```

Description

This function creates an I/O request item.

When you create an I/O request, you must decide how the device will notify you when an I/O operation completes. There are two choices:

- Notification by signal
- Notification by message

With notification by signal, the device will send your task the SIGF_IODONE signal whenever an I/O operation completes. This is a low-overhead mechanism, which is also low on information. When you get the signal, all you know is that an I/O operation has completed. You do not know which operation has completed. This has to be determined by looking at the state of all outstanding I/O requests.

Notification by message involves slightly more overhead, but provides much more information. When you create the I/O request, you indicate a message port. Whenever an I/O operation completes, the device will send a message to that message port. The message will contain the following information:

msg_Result

Contains the io_Error value from the I/O request. This indicates the state of the I/O operation, whether it worked or failed.

msg_Val1

Contains the item number of the I/O request that completed.

msg_Val2

Contains the value of the ioi_UserData field taken from the IOInfo structure used when initiating the I/O operation.

You can also create I/O requests which send you different signals when they complete. You must use CreateItem() and build up the tag list yourself to do this however. See the documentation entry for the IOReq item for more information on this.

Arguments

name

The name of the I/O request, or NULL if unnamed.

pri

The priority of the I/O request. For some devices, this value determines the order in which I/O requests are processed. When in doubt about what value to use, use 0.

dev

The item number of the device to which to send the I/O request. This device must have been opened by the current task or thread.

msgPort

If you want to receive a message when an I/O request is finished, this argument must be the item number of the message port to receive the message. To receive a signal instead, pass 0 for this argument.

Return Value

Returns the item number of the new I/O request, or a negative error code for failure. Possible error codes currently include:

BADITEM

The msgPort argument was not zero but did not specify a valid message port, or the device item didn't refer to a device.

ER_Kr_ItemNotOpen

The device specified by the dev argument is not open.

NOMEM

There was not enough memory to complete the operation.

Implementation

Link library call implemented in libc.a V20.

Associated Files

<:kernel:io.h>, libc.a

See Also

DeleteIOReq()

DeleteIOReq

Deletes an I/O request.

Synopsis

```
Err DeleteIOReq(Item ior);
```

Description

This macro deletes an I/O request item. You can use this macro in place of `DeleteItem()` to delete the item. If there was any outstanding I/O with this IOReq, it will be aborted first.

Arguments

`ior`
The item number of the I/O request to delete.

Return Value

Returns 0 if the I/O request was successfully deleted or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:io.h>` V20.

Associated Files

`<:kernel:io.h>`, `libc.a`

See Also

`CreateIOReq()`

DoIO

Performs synchronous I/O.

Synopsis

```
Err DoIO(Item ior, const IOInfo *ioiP);
```

Description

This function is the most efficient way to perform synchronous I/O. It works like `SendIO()`, except that it guarantees that the I/O operation has been completed before returning.

Arguments

`ior`

The item number of the I/O request to use.

`ioiP`

A pointer to the `IOInfo` structure containing the I/O command to be executed, input and/or output data, and other information.

Return Value

Returns ≥ 0 when the IO operation was successful, or a negative error code for failure. This value will be equal to the `IOReq.io_Error` field.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

`<:kernel:io.h>`, `libc.a`

See Also

`AbortIO()`, `CheckIO()`, `CreateIOReq()`, `DeleteIOReq()`, `SendIO()`, `WaitIO()`

SendIO

Requests I/O be performed.

Synopsis

```
Err SendIO(Item ior, const IOInfo *ioiP);
```

Description

This function sends an I/O request to a device to initiate an I/O operation.

Call this function after creating the necessary IOReq item (for the ior argument, created by calling `CreateIOReq()`) and an IOInfo structure (for the ioiP argument). The IOReq item specifies the device to which to send the request and the way your task will be notified when the I/O operation is completed. The IOInfo structure you supply specifies the parameters for the I/O operation.

This function is essentially asynchronous in nature. The I/O request is given to the device, and the device indicates completion of the operation by sending your task a signal or a message.

In many cases, it is possible for an I/O operation to be completed very quickly. In such a case, the overhead of sending your task a signal or a message to indicate that the I/O operation is completed can get to be a burden. This is the reason for the `IO_QUICK` flag.

If you set the `IO_QUICK` flag in the `ioi_Flags` field of the IOInfo structure, it indicates that you are prepared to deal with synchronous command completion. If the device is able to complete the I/O operation "quickly", then `SendIO()` will return 1, and no further notification will be sent to your task about the I/O completion (no signal or message).

If you specify `IO_QUICK`, it doesn't mean you'll get it however, it is merely a way for you to tell the system you are prepared to handle synchronous completion if it is possible.

When `SendIO()` returns 0, it means that the I/O operation is being handled asynchronously, and you will receive notification via signal or message when the I/O completes.

If you insist on synchronous completion, you should use `DoIO()` instead of `SendIO()`.

The IOInfo structure must be fully initialized before calling this function. You can use the `ioi_UserData` field of the IOInfo structure to contain whatever you want. This is a useful place to store a pointer to contextual data that needs to be associated with the I/O request. If you use message-based notification for your I/O requests, the `msg_Val2` field of the notification messages will contain the value of `ioi_UserData` from the IOInfo structure.

When using message-based notification, you must be sure to remove the notification messages from your message port between calls to `SendIO()`.

Arguments

ior

The item number of the IOReq structure for the request. This structure is normally created by calling `CreateIOReq()`.

ioiP

A pointer to a fully initialized IOInfo structure.

Return Value

Returns 1 if the I/O was completed immediately and no further notification will be sent. Returns 0 if the

I/O is being completed asynchronously, which means a signal or message will be sent to your task when the I/O completes. Returns a negative error code for failure. Possible error codes currently include:

BADITEM

The `ior` argument does not specify a valid `IOReq`.

NOTOWNER

The I/O request specified by the `ior` argument is not owned by the calling task.

ER_IONotDone

The I/O request is already in use for another I/O operation.

BADPTR

A pointer is invalid: Either the `IOInfo` structure specified by the `ioiP` argument is not entirely within the task's memory, the `IOInfo` receive buffer (specified by the `ioi_Recv` field in the `IOInfo` structure) is not entirely within the task's memory, or the `IOInfo` send buffer (specified by the `ioi_Send` field in the `IOInfo` structure) is not in legal memory.

BADCOMMAND

The device the I/O request is being sent to doesn't implement the command specified in the `ioi_Command` field of the `IOInfo` structure.

BADIOARG

Illegal arguments were given in the `IOInfo` structure for the given command.

MSGSENT

An attempt was made to start an I/O operation without having removed the completion message of a previous I/O operation involving this I/O request from the message port it is on.

When `SendIO()` returns 0 to indicate asynchronous completion, then the `IOReq`'s `io_Error` field will contain a result code for the I/O operation. Like all Portfolio error codes, a negative value indicates failure, and ≥ 0 indicates success. in `io_Error` of the `IOReq` structure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:io.h>`, `libc.a`

See Also

`AbortIO()`, `CheckIO()`, `DoIO()`, `WaitIO()`

WaitIO

Waits for an I/O operation to complete.

Synopsis

```
Err WaitIO(Item ior);
```

Description

The function puts the calling task into wait state until the specified I/O request signals completion. When a task is in wait state, it uses no CPU time.

If the I/O request has already completed, the function returns immediately. If the I/O request never completes and it is not aborted, the function never returns.

Arguments

ior

The item number of the I/O request to wait for.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in kernel folio V24.

Associated Files

`<:kernel:io.h>`, `libc.a`

See Also

`AbortIO()`, `CheckIO()`, `DoIO()`, `SendIO()`

CheckItem

Checks to see if an item exists.

Synopsis

```
void *CheckItem( Item i, uint8 ftype, uint8 ntype )
```

Description

This function checks to see if a given item exists. To specify the item, you use an item number, an item-type number, and the item number of the folio in which the item type is defined. If all three of these values match those of the item, `CheckItem()` returns a pointer to the item.

Arguments

`i`

The item number of the item to be checked.

`ftype`

The item number of the folio that defines the item type. (This is the same value that is passed as first parameter to the `MkNodeID()` macro.)

`ntype`

The item-type number for the item. (This is the same value that is passed as second parameter to the `MkNodeID()` macro.)

Return Value

If the item exists (and the values of all three arguments match those of the item), this function returns a pointer to the item. If the item does not exist, it returns `NULL`.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:item.h>`, `libc.a`

See Also

`CreateItem()`, `FindItem()`, `FindNamedItem()`, `LookupItem()`, `MkNodeID()`

CloseItem

Closes a previously opened item.

Synopsis

```
Err CloseItem( Item it )
```

Description

System items are created automatically by the operating system, such as folios and devices, and thus do not need to be created by tasks. To use a system item, a task first opens it by calling `OpenItem()`. When a task finishes using a system item, it calls `CloseItem()` to inform the operating system that it is no longer using the item.

Arguments

it
Number of the item to close.

Return Value

≥ 0 if the item was closed successfully or a negative error code for failure. Possible error codes currently include:

BADITEM
The i argument is not an item.

ER_Kr_ItemNotOpen
The i argument is not an opened item.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

<:kernel:item.h>, libc.a

See Also

`OpenItem()`

CreateItem

Creates an item.

Synopsis

```
Item CreateItem( int32 ct, TagArg *p )
Item CreateItemVA( int32 ct, uint32 tags, ... );
```

Description

This function creates an item.

There are convenience routines to create many types of items (such as `CreateMsg()` to create a message and `CreateIOReq()` to create an I/O request). You should use `CreateItem()` only when no convenience routine is available or when you need to supply additional arguments for the creation of the item beyond what the convenience routine provides.

When you no longer need an item created with `CreateItem()`, use `DeleteItem()` to delete it.

Arguments

ct

Specifies the type of item to create. Use `MkNodeID()` to generate this value.

p

A pointer to an array of tag arguments. The tag arguments can be in any order. The last element of the array must be the value `TAG_END`. If there are no tag arguments, this argument must be `NULL`.

Return Value

Returns the item number of the new item or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:item.h>`, `libc.a`

See Also

`CheckItem()`, `CreateIOReq()`, `CreateMsg()`, `CreateMsgPort()`,
`CreateSemaphore()`, `CreateSmallMsg()`, `CreateThread()`, `DeleteItem()`

DeleteItem

Deletes an item.

Synopsis

```
Err DeleteItem( Item it )
```

Description

This function deletes the specified item and frees any resources (including memory) that were allocated for the item.

There are convenience procedures for deleting most types of items (such as `DeleteMsg()` to delete a message and `DeleteIOReq()` to delete an I/O request). You should use `DeleteItem()` only if you used `CreateItem()` to create the item. If you used a convenience routine to create an item, you must use the corresponding convenience routine to delete the item.

If a task tries to perform operations on an item that has been deleted, the system will return an error code indicating the item doesn't exist. Note that it is possible for item numbers to be reused. This generally only happens after the system has been up and running without a reboot for a few weeks. This can become an issue in set-top box environments.

Tasks can only delete items that they own. If a task transfers ownership of an item to another task, it can no longer delete the item. The lone exception to this is the task itself; you can always commit suicide regardless of who owns the task/thread item.

When a task dies, the kernel automatically deletes all of the items it has created, and closes any items it has opened. In addition, any semaphores the task has locked get unlocked, and any messages the task hasn't replied to get replied with an error code.

Arguments

it
Number of the item to be deleted.

Return Value

≥ 0 if successful or a negative error code for failure.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

`<:kernel:item.h>`, `libc.a`

See Also

`CheckItem()`, `CreateItem()`, `DeleteIOReq()`, `DeleteMsg()`, `DeleteMsgPort()`, `DeleteThread()`, `exit()`

FindAndOpenItem

Finds an item by type and tags and opens it.

Synopsis

```
Item FindAndOpenItem( int32 cType, TagArg *tp )  
Item FindAndOpenItemVA( int32 cType, uint32 tags, ... );
```

Description

This function finds an item of the specified type whose tag arguments match those pointed to by the `tp` argument. If more than one item of the specified type has matching tag arguments, this function returns the item number for the first matching item. When an item is found, it is automatically opened and prepared for use.

Arguments

`cType`

Specifies the type of the item to find. Use `MkNodeID()` to generate this value.

`tp`

A pointer to an array of tag arguments. The tag arguments can be in any order. The array can contain some, all, or none of the possible tag arguments for an item of the specified type; to make the search more specific, include more tag arguments. The last element of the array must be the value `TAG_END`. If there are no tag arguments, this argument must be `NULL`.

Return Value

Returns the number of the first item that matches or a negative error code if it can't find the item or if an error occurs. The returned item is already opened, so there is no need to call `OpenItem()` on it. You should call `CloseItem()` on the supplied item when you are done with it.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

`<:kernel:item.h>`, `libc.a`

See Also

`CheckItem()`, `FindNamedItem()`, `LookupItem()`, `MkNodeID()`, `OpenItem()`, `FindItem()`

FindAndOpenNamedItem

Finds an item by name and opens it.

Synopsis

```
Item FindAndOpenNamedItem(int32 ctype, const char *name);
```

Description

This function finds an item of the specified type and name. The search is not case-sensitive. When an item is found, it is opened and prepared for use.

Arguments

ctype
The type of the item to find. Use `MkNodeID()` to create this value.

name
The name of the item to find.

Return Value

Returns the number of the item that was opened, or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:item.h>`, `libc.a`

See Also

`FindAndOpenItem()`, `CloseItem()`

FindItem

Finds an item by type and tags.

Synopsis

```
Item FindItem( int32 cType, TagArg *tp )  
Item FindItemVA( int32 cType, uint32 tags, ...)
```

Description

This function finds an item of the specified type whose tag arguments match those pointed to by the tp argument. If more than one item of the specified type has matching tag arguments, this function returns the item number for the first matching item.

Arguments

cType

Specifies the type of the item to find. Use MkNodeID() to generate this value.

tp

A pointer to an array of tag arguments. The tag arguments can be in any order. The array can contain some, all, or none of the possible tag arguments for an item of the specified type; to make the search more specific, include more tag arguments. The last element of the array must be the value TAG_END. If there are no tag arguments, this argument must be NULL.

Return Value

Returns the number of the first item that matches or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:item.h>, libc.a

Caveats

This function currently ignores most tag arguments.

See Also

CheckItem(), FindNamedItem(), LookupItem(), MkNodeID()

FindNamedItem

Finds an item by name.

Synopsis

```
Item FindNamedItem(int32 ctype, const char *name);
```

Description

This function finds an item of the specified type and name. The search is not case-sensitive.

Arguments

ctype

The type of the item to find. Use `MkNodeID()` to create this value.

name

The name of the item to find.

Return Value

Returns the item number of the item that was found, or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

`<:kernel:item.h>`, `libc.a`

See Also

`FindItem()`

IsItemOpened

Determines whether a task or thread has opened a given item.

Synopsis

```
Err IsItemOpened( Item task, Item it )
```

Description

This function determines whether a task or thread has currently got an item opened.

Arguments

task

The task or thread to inquire about. For the current task, use CURRENTTASKITEM.

it

The number of the item to verify.

Return Value

≥ 0 if the item was opened, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:item.h>, libc.a

See Also

OpenItem(), CloseItem(), CheckItem(), LookupItem()

LookupItem

Gets a pointer to an item.

Synopsis

```
void *LookupItem( Item it )
```

Description

This function finds an item by its item number and returns a pointer to the item.

Note: Because items are owned by the system, user tasks cannot change the values of their fields. They can only read the values contained in the public fields.

Arguments

it
The number of the item to look up.

Return Value

Returns a pointer to the item or NULL if the item does not exist.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:item.h>, libc.a

See Also

CheckItem(), CreateItem(), FindItem(), FindNamedItem()

MkNodeID

Assembles an item type value.

Synopsis

```
int32 MkNodeID( uint8 ftype , uint8 nType )
```

Description

This macro creates an item type value, a 32-bit value that specifies an item type and the folio in which the item type is defined. This value is required by other functions that deal with items, such as `CreateItem()` and `FindItem()`.

Arguments

ftype

The item number of the folio in which the item type of the item is defined.

ntype

The item type number for the item.

Return Value

Returns an item type value, useful for such calls as `CreateItem()`.

Implementation

Macro implemented in `<:kernel:nodes.h>` V20.

Associated Files

`<:kernel:nodes.h>`, `libc.a`

See Also

`CreateItem()`, `FindItem()`, `FindNamedItem()`

OpenItem

Opens an item.

Synopsis

```
Item OpenItem( Item FoundItem, void *args )
```

Description

This function opens an item for access.

System items are created automatically by the operating system, such as folios and devices, and thus do not need to be created by tasks. To use a system item, a task first opens it by calling `OpenItem()`. When a task finishes using a system item, it calls `CloseItem()` to inform the operating system that it is no longer using the item.

Arguments

item

The number of the item to open. To find the item number of a system item use `FindItem()` or `FindNamedItem()`.

args

A pointer to an array of tag arguments. Currently, this must be `NULL`.

Return Value

Returns the number of the item that was opened or a negative error code for failure. The item number for the opened item is not always the same as the item number returned by `FindItem()`. When accessing the opened item you should use the return from `OpenItem()`, not the return from `FindItem()`. You can also call the `FindAndOpenItem()` function to do both a search and an open operation in an atomic manner.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:item.h>, libc.a

See Also

`CloseItem()`, `FindItem()`

SetItemOwner

Changes the owner of an item.

Synopsis

```
Err SetItemOwner( Item i, Item newOwner )
```

Description

This function makes another task the owner of an item. A task must be the owner of the item to change its ownership. The item is removed from the current task's resource table and placed in the new owner's resource table.

Arguments

i
The item number of the item to give to a new owner.

newOwner
The item number of the new owner task.

Return Value

>= 0 if ownership of the item is changed or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:item.h>, libc.a

Caveats

This is not currently implemented for all items that should be able to have their ownership changed.

See Also

CreateItem(), FindItem(), DeleteItem()

SetItemPri

Changes the priority of an item.

Synopsis

```
int32 SetItemPri( Item i, uint8 newpri )
```

Description

This function changes the priority of an item. Some items in the Portfolio are maintained in lists; for example, tasks and threads. Some lists are ordered by priority, with higher-priority items coming before lower-priority items. When the priority of an item in a list changes, the kernel automatically rearranges the list of items to reflect the new priority. The item is moved immediately before the first item whose priority is lower.

A task must own an item to change its priority. A task can change its own priority even if it does not own itself.

Arguments

i

The item number of the item whose priority to change.

newpri

The new priority for the item.

Return Value

Returns the previous priority of the item or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:item.h>, libc.a

Notes

For certain item types, such as devices, the kernel may change the priority of an item to help optimize throughput.

Caveats

This function is currently not implemented for many item types.

See Also

CreateItem()

AddHead

Adds a node to the head of a list.

Synopsis

```
void AddHead( List *l, Node *n )
```

Description

This function adds a node to the head (the beginning) of the specified list.

Arguments

- l
A pointer to the list in which to add the node.
- n
A pointer to the node to add.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

Caveats

Attempting to insert a node into a list while it is a member of another list is not reported as an error, and will trash the other list.

See Also

```
AddTail(),  PrepList(),  InsertNodeFromHead(),  InsertNodeFromTail(),  
RemHead(),   RemNode(),   RemTail(),   UniversalInsertNode(),  
InsertNodeAlpha()
```

AddTail

Adds a node to the tail of a list.

Synopsis

```
void AddTail( List *l, Node *n )
```

Description

This function adds the specified node to the tail (the end) of the specified list.

Arguments

l

A pointer to the list in which to add the node.

n

A pointer to the node to add.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

Caveats

Attempting to insert a node into a list while it is a member of another list is not reported as an error, and will trash the other list.

See Also

AddHead(), PrepList(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail(), UniversalInsertNode(),
InsertNodeAlpha()

DumpNode

Prints contents of a node.

Synopsis

```
void DumpNode (const Node *node, const char *banner)
```

Description

This function prints out the contents of a Node structure for debugging purposes.

Arguments

node

The node to print.

banner

Descriptive text to print before the node contents. May be NULL.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:list.h>, libc.a

FindNamedNode

Finds a node by name.

Synopsis

```
Node *FindNamedNode( const List *l, const char *name )
```

Description

This function searches a list for a node with the specified name. The search is not case-sensitive (that is, the kernel does not distinguish uppercase and lowercase letters in node names).

Arguments

l

A pointer to the list to search.

name

The name of the node to find.

Return Value

The function returns a pointer to the node structure or NULL if the named node couldn't be found.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

See Also

FirstNode(), LastNode(), NextNode(), PrevNode()

FindNodeFromHead

Returns a pointer to a node appearing at a given ordinal position from the head of the list.

Synopsis

```
Node *FindNodeFromHead(const List *l, uint32 position);
```

Description

This function scans the supplied list and returns a pointer to the node appearing in the list at the given ordinal position. NULL is returned if the list doesn't contain that many items.

Arguments

l

A pointer to the list to scan for the node.

position

The node position to look for.

Return Value

A pointer to the node found, or NULL if the list doesn't contain enough nodes.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:list.h>, libc.a

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail()

FindNodeFromTail

Returns a pointer to a node appearing at a given ordinal position from the tail of the list.

Synopsis

```
Node *FindNodeFromTail(const List *l, uint32 position);
```

Description

This function scans the supplied list and returns a pointer to the node appearing in the list at the given ordinal position counting from the end of the list. NULL is returned if the list doesn't contain that many items.

Arguments

l

A pointer to the list to scan for the node.

position

The node position to look for, relative to the end of the list.

Return Value

A pointer to the node found, or NULL if the list doesn't contain enough nodes.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:list.h>, libc.a

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail()

FirstNode

Gets the first node in a list.

Synopsis

```
Node *FirstNode( List *l )
```

Description

This macro returns a pointer to the first node in the specified list. If the list is empty, the macro returns a pointer to the tail (end-of-list) anchor. To determine if the return value is an actual node rather than the tail anchor, call the `IsNode()` function.

Arguments

1

A pointer to the list from which to get the first node.

Return Value

The macro returns a pointer to first node in the list or, if the list is empty, a pointer to the tail (end-of-list) anchor.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Example

```
for (n = FirstNode(list); IsNode(list, n); n = NextNode(n))
{
    // n will iteratively point to every node in the list
}
```

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`IsListEmpty()`, `IsNode()`, `IsNodeB()`, `LastNode()`, `NextNode()`, `PrevNode()`, `ScanList()`

GetNodeCount

Counts the number of nodes in a list.

Synopsis

```
uint32 GetNodeCount(const List *l);
```

Description

This function counts the number of nodes currently in the list.

Arguments

l

A pointer to the list to count the nodes of.

Return Value

The number of nodes in the list.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<kernel:list.h>, libc.a

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail()

GetNodePosFromHead

Gets the ordinal position of a node within a list, counting from the head of the list.

Synopsis

```
int32 GetNodePosFromHead(const List *l, const Node *n);
```

Description

This function scans the supplied list looking for the supplied node, and returns the ordinal position of the node within the list. If the node doesn't appear in the list, the function returns -1.

Arguments

- l
A pointer to the list to scan for the node.
- n
A pointer to a node to locate in the list.

Return Value

The ordinal position of the node within the list counting from the head of the list, or -1 if the node isn't in the list. The first node in the list has position 0, the second node has position 1, etc.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:list.h>, libc.a

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail()

GetNodePosFromTail

Gets the ordinal position of a node within a list, counting from the tail of the list.

Synopsis

```
int32 GetNodePosFromTail(const List *l, const Node *n);
```

Description

This function scans the supplied list backward looking for the supplied node, and returns the ordinal position of the node within the list. If the node doesn't appear in the list, the function returns -1.

Arguments

l

A pointer to the list to scan for the node.

n

A pointer to a node to locate in the list.

Return Value

The ordinal position of the node within the list counting from the tail of the list, or -1 if the node isn't in the list. The last node in the list has position 0, the second to last node has position 1, etc.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:list.h>, libc.a

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail()

InsertNodeAfter

Inserts a node into a list after another node already in the list.

Synopsis

```
void InsertNodeAfter(Node *oldNode, Node *newNode);
```

Description

This function lets you insert a new node into a list, AFTER another node that is already in the list.

Arguments

`oldNode`
The node after which to insert the new node.

`newNode`
The node to insert in the list.

Implementation

Link library call implemented in libc.a V24.

Associated Files

`<:kernel:list.h>`, libc.a

Notes

A node can be included only in one list.

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromHead()`, `InsertNodeFromTail()`,
`RemHead()`, `RemNode()`, `RemTail()`, `InsertNodeBefore()`

InsertNodeAlpha

Inserts a node into a list according to alphabetical order.

Synopsis

```
void InsertNodeAlpha(List *l, Node *n);
```

Description

This function lets you insert a new node into a list, according to alphabetical order of the name of the node. The list is assumed to already be sorted in alphabetical order.

Arguments

l

A pointer to the list into which to insert the node.

n

A pointer to the node to insert.

Implementation

Link library call implemented in libc.a V27.

Associated Files

<:kernel:list.h>, libc.a

Notes

A node can be included only in one list.

See Also

AddHead(), AddTail(), InsertNodeFromHead(), InsertNodeFromTail(),
RemHead(), RemNode(), RemTail(), InsertNodeAfter(), InsertNodeBefore()

InsertNodeBefore

Inserts a node into a list before another node already in the list.

Synopsis

```
void InsertNodeBefore(Node *oldNode, Node *newNode);
```

Description

This function lets you insert a new node into a list, BEFORE another node that is already in the list.

Arguments

`oldNode`
The node before which to insert the new node.

`newNode`
The node to insert in the list.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:list.h>, libc.a

Notes

A node can be included only in one list.

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromHead()`, `InsertNodeFromTail()`,
`RemHead()`, `RemNode()`, `RemTail()`, `InsertNodeAfter()`

InsertNodeFromHead

Inserts a node into a list.

Synopsis

```
void InsertNodeFromHead( List *l, Node *n )
```

Description

This function inserts a new node into a list. The order of nodes in a list is often determined by their priority. The function compares the priority of the new node to the priorities of nodes currently in the list, beginning at the head of the list, and inserts the new node immediately after all nodes whose priority is higher. If the priorities of all the nodes in the list are higher, the node is added at the end of the list.

To arrange the nodes in a list by a value or values other than priority, use `UniversalInsertNode()`.

Arguments

`l`

A pointer to the list into which to insert the node.

`n`

A pointer to the node to insert.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`AddHead()`, `AddTail()`, `RemHead()`, `RemNode()`, `RemTail()`,
`InsertNodeFromTail()`, `UniversalInsertNode()`, `InsertNodeBefore()`,
`InsertNodeAfter()`

InsertNodeFromTail

Inserts a node into a list.

Synopsis

```
void InsertNodeFromTail( List *l, Node *n )
```

Description

This function inserts a new node into a list. The order of nodes in a list is often determined by their priority. The function compares the priority of the new node to the priorities of nodes currently in the list, beginning at the tail of the list, and inserts the new node immediately before the nodes whose priority is lower. If there are no nodes in the list whose priority is lower, the node is added at the head of the list.

To arrange the nodes in a list by a value or values other than priority, use `UniversalInsertNode()`.

Arguments

l

A pointer to the list into which to insert the node.

n

A pointer to the node to insert.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

See Also

```
AddHead(),      AddTail(),      RemHead(),      RemNode(),      RemTail(),  
InsertNodeFromHead(),  UniversalInsertNode(),  InsertNodeBefore()  
InsertNodeAfter()
```

IsEmptyList

Checks whether a list is empty.

Synopsis

```
bool IsEmptyList( List *l )
```

Description

This macro checks whether a list is empty.

Arguments

1

A pointer to the list to check.

Return Value

The macro returns TRUE if the list is empty or FALSE if it isn't.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`FirstNode()`, `IsNode()`, `IsNodeB()`, `LastNode()`, `NextNode()`, `PrevNode()`,
`ScanList()`, `IsListEmpty()`

IsListEmpty

Checks whether a list is empty.

Synopsis

```
bool IsListEmpty( List *l )
```

Description

This macro checks whether a list is empty.

Arguments

l

A pointer to the list to check.

Return Value

The macro returns TRUE if the list is empty or FALSE if it isn't.

Implementation

Macro implemented in `<:kernel:list.h>` V24.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`FirstNode()`, `IsNode()`, `IsNodeB()`, `LastNode()`, `NextNode()`, `PrevNode()`,
`ScanList()`

IsNode

Validates a node when moving forward through a list.

Synopsis

```
bool IsNode( const List *l, const Node *n )
```

Description

This macro is used to test whether the specified node is an actual node or is the tail (end-of-list) anchor. Use this macro when traversing a list from head to tail. When traversing a list from tail to head, use the `IsNodeB()` macro.

Arguments

`l`

A pointer to the list containing the node to check.

`n`

A pointer to the node to check.

Return Value

The macro returns `TRUE` if the node is an actual node or `FALSE` if it is the tail (end-of-list) anchor. This macro will return `TRUE` for any node that is not the tail anchor, whether or not the node is in the specified list.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`FirstNode()`, `IsListEmpty()`, `IsNodeB()`, `LastNode()`, `NextNode()`, `PrevNode()`, `ScanList()`

IsNodeB

Validates a node when moving backward through a list.

Synopsis

```
bool IsNodeB( const List *l, const Node *n )
```

Description

This macro is used to test whether the specified node is an actual node or is the head (beginning-of-list) anchor. Use this macro when traversing a list from tail to head. When traversing a list from head to tail, use the `IsNode()` macro.

Arguments

l

A pointer to the list containing the node to check.

n

A pointer to the node to check.

Return Value

The macro returns `TRUE` if the node is an actual node or `FALSE` if it is the head (beginning-of-list) anchor. This macro will return `TRUE` for any node that is not the head anchor, whether or not the node is in the specified list.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`FirstNode()`, `IsListEmpty()`, `IsNode()`, `LastName()`, `NextNode()`, `PrevNode()`, `ScanListB()`

LastNode

Gets the last node in a list.

Synopsis

```
Node *LastNode( const List *l )
```

Description

This macro returns a pointer to the last node in a list. If the list is empty, the macro returns a pointer to the head (beginning-of-list) anchor. To determine if the return value is an actual node rather than the head anchor, call the `IsNodeB()` function.

Arguments

1

A pointer to the list structure to be examined.

Return Value

The macro returns a pointer to last node in the list or, if the list is empty, a pointer to the head (beginning-of-list) anchor.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Example

```
for (n = LastNode(list); IsNodeB(list, n); n = PrevNode(n))
{
    // n will iteratively point to every node in the list
}
```

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`FirstNode()`, `IsListEmpty()`, `IsNode()`, `IsNodeB()`, `NextNode()`, `PrevNode()`, `ScanList()`

NextNode

Gets the next node in a list.

Synopsis

```
Node *NextNode( const Node *n )
```

Description

This macro gets a pointer to the next node in a list. If the current node is the last node in the list, the result is a pointer to the tail (end-of-list) anchor. To determine if the return value is an actual node rather than the tail anchor, call the `IsNode()` function.

Arguments

`n`
Pointer to the current node.

Return Value

The macro returns a pointer to the next node in the list or, if the current node is the last node in the list, to the tail (end-of-list) anchor.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

Caveats

Assumes that `n` is a node in a list. If not, watch out.

See Also

`FirstNode()`, `IsListEmpty()`, `IsNode()`, `IsNodeB()`, `ListNode()`, `PrevNode()`, `ScanList()`

PrepList

Initializes a list.

Synopsis

```
void PrepList( List *l )
```

Description

When you create a List structure, you must initialize it with a call to `PrepList()` before using it. `PrepList()` creates an empty list by initializing the head (beginning-of-list) and tail (end-of-list) anchors.

You must initialize all List structures with `PrepList()` or `PREPLIST()` before using them with any other List function

Arguments

l

A pointer to the list to be initialized.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:list.h>, libc.a

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromHead()`, `InsertNodeFromTail()`, `IsEmptyList()`, `RemHead()`, `RemNode()`, `RemTail()`, `UniversalInsertNode()`

PrevNode

Gets the previous node in a list.

Synopsis

```
Node *PrevNode( const Node *node )
```

Description

This macro returns a pointer to the previous node in a list. If the current node is the first node in the list, the result is a pointer to the head (beginning-of-list) anchor. To determine whether the return value is an actual node rather than the head anchor, use the `IsNodeB()` function.

Arguments

node
A pointer to the current node.

Return Value

The macro returns a pointer to the previous node in the list or, if the current node is the first node in the list, to the head (beginning-of-list) anchor.

Implementation

Macro implemented in `<:kernel:list.h>` V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`FirstNode()`, `IsListEmpty()`, `IsNode()`, `IsNodeB()`, `LastNode()`, `ScanList()`

RemHead

Removes the first node from a list.

Synopsis

```
Node *RemHead( List *l )
```

Description

This function removes the head (first) node from a list.

In a development environment, the link fields of the node structure are modified so that an attempt to remove the same node a second time will cause your task to crash due to accessing non-existent memory. The address of this access gives you a bit of information about what happened. If the address is 0xABADC0DE or 0xABADFACE, it means that the node was first removed from the list by a call to `RemNode()`. If the address of the access is 0xAF00DBAD or 0xAFEEDBAD, the node was first removed from the list using `RemHead()`. And finally, if the address of the access is 0xABADF00D or 0xABADFEED, the node was first removed using `RemTail()`.

Arguments

1

A pointer to the list to be beheaded.

Return Value

The function returns a pointer to the node that was removed from the list or NULL if the list was empty.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromTail()`, `RemNode()`, `RemTail()`

RemNode

Removes a node from a list.

Synopsis

```
void RemNode( Node *n )
```

Description

This function removes the specified node from a list.

In a development environment, the link fields of the node structure are modified so that an attempt to remove the same node a second time will cause your task to crash due to accessing non-existent memory. The address of this access gives you a bit of information about what happened. If the address is 0xABADC0DE or 0xABADFACE, it means that the node was first removed from the list by a call to `RemNode()`. If the address of the access is 0xAF00DBAD or 0xAFEEADBAD, the node was first removed from the list using `RemHead()`. And finally, if the address of the access is 0xABADF00D or 0xABADFEED, the node was first removed using `RemTail()`.

Arguments

n

A pointer to the node to remove.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromTail()`, `RemHead()`, `RemTail()`

RemTail

Removes the last node from a list.

Synopsis

```
Node *RemTail( List *l )
```

Description

This function removes the tail (last) node from a list.

In a development environment, the link fields of the node structure are modified so that an attempt to remove the same node a second time will cause your task to crash due to accessing non-existent memory. The address of this access gives you a bit of information about what happened. If the address is 0xABADC0DE or 0xABADFACE, it means that the node was first removed from the list by a call to `RemNode()`. If the address of the access is 0xAF00DBAD or 0xAFEEDBAD, the node was first removed from the list using `RemHead()`. And finally, if the address of the access is 0xABADF00D or 0xABADFEEED, the node was first removed using `RemTail()`.

Arguments

1

A pointer to the list to have it tail removed.

Return Value

The function returns a pointer to the node that was removed from the list or NULL if the list was empty.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromTail()`, `RemHead()`, `RemNode()`

ScanList

Walks through all the nodes in a list.

Synopsis

```
ScanList( const List *l, void *n, <node type>)
```

Description

This macro lets you easily walk through all the elements in a list from the first to the last.

Arguments

- l**
A pointer to the list to scan.
- n**
A variable which will be altered to hold a pointer to every node in the list in succession.
- <node type>**
The data type of the nodes in the list. This is used for type casting within the macro.

Implementation

Macro implemented in `<:kernel:list.h>` V22.

Example

```
{
List          *l;
DataStruct    *d;
uint32        i;

    i = 0;
    ScanList(l,d,DataStruct)
    {
        printf("Node %d is called %sn",i,d->d.n_Name);
        i++;
    }
}
```

Associated Files

`<:kernel:list.h>`, `libc.a`

Warning

You cannot remove nodes from the list as you are scanning it with this macro.

See Also

`ScanListB()`

ScanListB

Walks through all the nodes in a list backwards.

Synopsis

```
ScanListB( const List *l, void *n, <node type>)
```

Description

This macro lets you easily walk through all the elements in a list from the last to the first.

Arguments

- l**
A pointer to the list to scan.
- n**
A variable which will be altered to hold a pointer to every node in the list in succession.
- <node type>**
The data type of the nodes on the list. This is used for type casting within the macro.

Implementation

Macro implemented in `<:kernel:list.h>` V24.

Example

```
{
List      *l;
DataStruct *d;
uint32    i;

    i = 0;
    ScanListB(l,d,DataStruct)
    {
        printf("Node %d (counting from the end) is called %sn",i,
               d->d.n_Name);
        i++;
    }
}
```

Associated Files

`<:kernel:list.h>`, `libc.a`

Warning

You cannot remove nodes from the list as you are scanning it with this macro.

See Also

`ScanList()`

SetNodePri

Changes the priority of a list node.

Synopsis

```
uint8 SetNodePri( Node *n, uint8 newpri )
```

Description

This function changes the priority of a node in a list. The kernel arranges lists by priority, with higher-priority nodes coming before lower-priority nodes. When the priority of a node changes, the kernel automatically rearranges the list to reflect the new priority. The node is moved immediately before the first node whose priority is lower.

Arguments

n
A pointer to the node whose priority to change.

newpri
The new priority for the node.

Return Value

The function returns the previous priority of the node.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:list.h>, libc.a

Notes

If you know the list that contains the node you wish to change the priority of, it is generally much faster to do:

```
RemNode(n);  
n->n_Priority = newPri;  
InsertNodeFromTail(l,n);
```

See Also

```
InsertNodeFromHead(), InsertNodeFromTail(), UniversalInsertNode()
```

UniversalInsertNode

Inserts a node into a list.

Synopsis

```
void UniversalInsertNode( List *l, Node *n, bool (*f)(Node *n,Node *m) )
```

Description

Every node in a list has a priority (a value from 0 to 255 that is stored in the `n_Priority` field of the node structure). When a new node is inserted with `InsertNodeFromHead()` or `InsertNodeFromTail()`, the position at which it is added is determined by its priority. In contrast, the `UniversalInsertNode()` function allows you to arrange nodes according to values other than priority.

`UniversalInsertNode()` uses a comparison function provided by the calling task to determine where to insert a new node. It compares the node to be inserted with nodes already in the list, beginning with the first node. If the comparison function returns `TRUE`, the new node is inserted immediately before the node to which it was compared. If the comparison function never returns `TRUE`, the new node becomes the last node in the list. The comparison function, whose arguments are pointers to two nodes, can use any data in the nodes for the comparison.

Arguments

`l`

A pointer to the list into which to insert the node.

`n`

A pointer to the node to insert. This same pointer is passed as the first argument to the comparison function.

`f`

A comparison function provided by the calling task that returns `TRUE` if the node to be inserted (pointed to by the first argument to the function) should be inserted immediately before the node to which it is compared (pointed to by the second argument to the function).

`m`

A pointer to the node in the list to which to compare the node to insert.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:list.h>`, `libc.a`

See Also

`AddHead()`, `AddTail()`, `InsertNodeFromHead()`, `InsertNodeFromTail()`, `RemHead()`, `RemNode()`, `RemTail()`, `InsertNodeBefore()`, `InsertNodeAfter()`

CloseModule

Concludes use of a module item.

Synopsis

```
Err CloseModule(Item module);
```

Description

Closes a module item previously opened with `OpenModule()`. Once the module is closed, it can no longer be used, as the system may unload it from memory.

Arguments

module

The module's item number, as obtained from `OpenModule()`.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

BADITEM

The supplied module argument is not a valid item.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:loader:loader3do.h>`, `libc.a`

See Also

`OpenModule()`, `ExecuteModule()`, `CreateModuleThread()`

ExecuteModule

Executes code in a module item as a subroutine.

Synopsis

```
int32 ExecuteModule(Item module, uint32 argc, char **argv);
```

Description

This function lets you execute a chunk of code that was previously loaded from disk using `OpenModule()`. The code will run as a subroutine of the current task or thread.

The `argc` and `argv` parameters are passed directly to the `main()` entry point of the loaded code. The return value of this function is the value returned by `main()` of the code being run.

The values you supply for `argc` and `argv` are irrelevant to this function. They are simply passed through to the loaded code. Therefore, their meaning must be agreed upon by the caller of this function, and by the loaded code.

Arguments

`module`

The item for the loaded code as obtained from `OpenModule()`.

`argc`

A value that is passed directly as the `argc` parameter to the loaded code's `main()` entry point. This function doesn't use the value of this argument, it is simply passed through to the loaded code.

`argv`

A value that is passed directly as the `argv` parameter to the loaded code's `main()` entry point. This function doesn't use the value of this argument, it is simply passed through to the loaded code.

Return Value

Returns the value that the loaded code's `main()` function returns, or `BADITEM` if the supplied module item is invalid.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

`OpenModule()`, `CloseModule()`, `CreateModuleThread()`

ExpungeByAddress

Removes a symbol from the list of exports

Synopsis

```
Err ExpungeByAddress( Item module, const void *address )
```

Description

This function removes a specified symbol from the exports table of a loaded module.

Arguments

module

The item number of the exporting module.

address

The address of the exporting symbol.

Return Value

Returns an error code.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

ExpungeBySymbol(), RedirectSymbol()

ExpungeBySymbol

Removes a symbol from the list of exports

Synopsis

```
Err ExpungeBySymbol( Item module, int32 symbolNumber )
```

Description

This function removes a specified symbol from the exports table of a loaded module.

Arguments

module

The item number of the exporting module.

symbolNumber

The symbol # to be expunged. Or one of the following special definitions:

SYM_ALL - Expunges all exports

SYM_DATA - Expunges all data exports

SYM_TEXT - Expunges all code exports

Return Value

Returns zero for success or one of the following error codes:

BADITEM

The module argument does not specify a module item. NOTOWNER

The module specified does not belong to this task. LOADER_ERR_IMPRANGE

The symbol number specified is out of bounds.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

ExpungeByAddress(), RedirectSymbol()

ImportByAddress

Loads an exporting module based on an imported symbol

Synopsis

```
Item ImportByAddress( Item module, void *address )
```

Description

This function loads an import by symbol passed by address.

Arguments

module
The item number of the importing module.

Return Value

Returns an an error code, or the module Item of the loaded module.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

ImportByName ()

ImportByName

Loads and resolves the named exporting module.

Synopsis

```
Item ImportByName( Item module, const char *name )
```

Description

Loads and resolves the named exporting module.

Arguments

module

The item number of the importing module.

name

The name of the exporting module.

Return Value

Returns the item module which was loaded, or an error code.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

ImportByAddress ()

LookupSymbol

Returns the address of a loaded symbol

Synopsis

```
void *LookupSymbol( Item module, int32 symbolNumber )
```

Description

This function returns the address of a exported symbol.

Arguments

module
The item number of the exporting module.

symbolNumber
The symbol # to be found.

Return Value

Returns an error code.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

OpenModule

Opens an executable file from disk and prepares it for use.

Synopsis

```
Item OpenModule(const char *path, OpenModuleTypes type,
               const TagArg *tags);
```

Description

This function loads an executable file from disk into memory. Once loaded, the code can be spawned as a thread, or executed as a subroutine.

Give this function the name of the executable file to load, and how to load it into memory. It builds a module item that represents the loaded code, and returns it to you.

Once you finish using the loaded code, you can remove it from memory by using `CloseModule()`.

To execute the loaded code, you can call either `ExecuteModule()`, `CreateModuleThread()`, or `CreateTask()`

Arguments

path
The file system pathname of the executable to open.

type
How the memory should be allocated for the module. If this value is `OPENMODULE_FOR_THREAD`, then the memory is allocated within the current task's pages. If this value is `OPENMODULE_FOR_TASK`, then the memory is allocated from new pages.

tags
This must currently be `NULL`.

Return Value

Returns the item number of the newly opened module, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:loader:loader3do.h>`, `libc.a`

See Also

`CloseModule()`, `ExecuteModule()`, `CreateModuleThread()`, `system()`

UnimportByAddress

Unloads an exporting module, based on the address of an import

Synopsis

```
Err UnimportByAddress( Item module, const void *address )
```

Description

This function unloads an exporting module, based on the address of an exported symbol.

Arguments

module
The item number of the importing module.

address
The address of a symbol.

Return Value

Returns an error code.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

UnimportByName()

UnimportByName

Unloads a named module

Synopsis

```
Err UnimportByName( Item module, const char *name )
```

Description

This function unloads the named module.

Arguments

module

The item number of the importing module.

name

The name of the exporting module.

Return Value

Returns an error code.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:loader:loader3do.h>, libc.a

See Also

UnimportByAddress()

ControlLumberjack

Determine which event types Lumberjack logs.

Synopsis

```
Err ControlLumberjack(uint32 eventsLogged);
```

Description

This function lets you control which events are logged by Lumberjack. After you call `CreateLumberjack()`, you must call this function in order to get logging to actually happen.

Arguments

`eventsLogged`

This is a bit mask that specifies which types of events should be logged by Lumberjack. See the `LOGF_CONTROL_XXX` constants in `<:kernel:lumberjack.h>` for the available types.

If you supply 0 for this argument, all logging will be stopped.

Return Value

`>= 0` for success, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

`<:kernel:lumberjack.h>`, `libc.a`

See Also

`ObtainLumberjackBuffer()`, `ReleaseLumberjackBuffer()`,
`DumpLumberjackBuffer()`, `CreateLumberjack()`, `LogEvent()`

CreateLumberjack

Initializes Lumberjack, the Portfolio logging service.

Synopsis

```
Err CreateLumberjack(const TagArg *tags);
```

Description

Lumberjack is used to log system events to aid during debugging. The kernel uses Lumberjack to store a large amount of information about what is currently happening in the system.

The information logged by the kernel includes all task switches, interrupts, signals or messages being sent, semaphores being locked and unlocked, pages of memory being allocated or freed, and more. You can also add your own events to the log using the `LogEvent()` function.

When you call this function, a list of buffers is allocated that currently total up to a little over 768K of memory. These buffers are used by Lumberjack to store the log entries. To start logging information in these buffers, you must call `ControlLumberjack()`.

As buffers get filled up, Lumberjack puts the buffers into a list of filled buffers. You can get the oldest buffer on this list by calling `ObtainLumberjackBuffer()`. Once you have the buffer, you can parse the log entries yourself to extract information, or you can call `DumpLumberjackBuffer()` to parse and output the contents of the buffer to the debugging terminal.

Arguments

tags

This is reserved for future use and must currently always be NULL.

Return Value

≥ 0 for success, or a negative error code for failure. The likely cause of failure is a lack of memory to allocate the needed buffer space.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

`<:kernel:lumberjack.h>`, `libc.a`

See Also

<code>ObtainLumberjackBuffer()</code> ,	<code>ReleaseLumberjackBuffer()</code> ,
<code>DumpLumberjackBuffer()</code> ,	<code>DeleteLumberjack()</code> ,
<code>ControlLumberjack()</code>	<code>LogEvent()</code> ,

DeleteLumberjack

Disables Lumberjack, the Portfolio logging service.

Synopsis

```
Err DeleteLumberjack(void);
```

Description

This function stops Lumberjack and releases all buffer space allocated by `CreateLumberjack()`. This causes any currently logged events to be discarded. See `ControlLumberjack()` for a way to stop logging without discarding the buffers.

Return Value

≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

`<:kernel:lumberjack.h>`, `libc.a`

See Also

<code>ObtainLumberjackBuffer()</code> ,	<code>ReleaseLumberjackBuffer()</code> ,
<code>DumpLumberjackBuffer()</code> ,	<code>CreateLumberjack()</code> ,
<code>ControlLumberjack()</code>	<code>LogEvent()</code> ,

DumpLumberjackBuffer

Parse and display the contents of a Lumberjack buffer.

Synopsis

```
void DumpLumberjackBuffer(const char *banner,  
                          const LumberjackBuffer *lb);
```

Description

Lumberjack, the Portfolio logging service, maintains a list of buffers into which it logs events during system execution.

This function takes a pointer to a LumberjackBuffer and decodes the information it contains, and displays the results to the debugging terminal. You obtain a pointer to a LumberjackBuffer by calling ObtainLumberjackBuffer().

Arguments

banner

A string to display before dumping the contents of the buffer. This may be NULL in which case no banner string will be printed.

lb

The buffer to dump the contents of, as obtained from ObtainLumberjackBuffer(). This may be NULL, in which case this function does nothing.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

<:kernel:lumberjack.h>, libc.a

See Also

ObtainLumberjackBuffer(),
ControlLumberjack(), LogEvent()

ReleaseLumberjackBuffer(),

LogEvent

Add a custom event to the Lumberjack logs.

Synopsis

```
Err LogEvent(const char *eventDescription);
```

Description

Lumberjack is used to log system events to aid during debugging. The kernel stores a large amount of information about what is currently happening in the system into Lumberjack. This function lets you add your own events to the logs, which can be useful to track high level events within your own code.

Arguments

eventDescription

This details your log entry. The event will be time stamped, marked with some default information such as the current task name, and inserted into the Lumberjack logs.

Return Value

>= 0 for success, or a negative error code for failure. The likely cause of failure is because you didn't call `CreateLumberjack()` to activate Lumberjack, or there is no more buffer space left to store the log entry. You can free up some buffer space by calling `ObtainLumberjackBuffer()` followed by `ReleaseLumberjackBuffer()`.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

<:kernel:lumberjack.h>, libc.a

See Also

<code>ObtainLumberjackBuffer()</code> ,	<code>ReleaseLumberjackBuffer()</code> ,
<code>DumpLumberjackBuffer()</code> ,	<code>CreateLumberjack()</code> ,
<code>ControlLumberjack()</code>	<code>DeleteLumberjack()</code> ,

ObtainLumberjackBuffer

Obtain a pointer to a Lumberjack buffer which is currently full.

Synopsis

```
LumberjackBuffer *ObtainLumberjackBuffer(void);
```

Description

Lumberjack, the Portfolio logging service, maintains a list of buffers into which it logs events during system execution.

This function returns a pointer to a logging buffer which is currently full of logged events. Once you have obtained such a buffer, you can parse its contents to extract useful information, or just call `DumpLumberjackBuffer()` to display the buffer contents to the debugging terminal.

Since Lumberjack maintains a list of buffers, when one buffer fills up, it simply gets a new buffer and starts logging events there. This scheme lets you call `ObtainLumberjackBuffer()` to get and parse a log buffer, while events are still being logged to a different buffer.

When you're done with a log buffer, you should return it to Lumberjack using the `ReleaseLumberjackBuffer()` function. If the buffer is not returned, it's possible Lumberjack will become unable to log events because it has no buffer space left.

Before you start obtaining buffers, it is generally a good idea to first disable event logging using `ControlLumberjack(0)`. Otherwise, new events will continuously be logged while you're dumping them or processing them, and you'll never see the end of it.

The `LumberjackBuffer.lb_BufferData` field points to the log entries for the buffer. Each log entry starts with a `LogEntryHeader` structure followed by a variable amount of data. The number of bytes before the next entry is stored in the `LogEntryHeader.leh_NextEvent` field. By adding this value to the current `LogEntryHeader` address, you get to the next `LogEntryHeader` structure in the buffer. The `LogEntryHeader.leh_Type` field indicates what type of event this is. If the type is `LOG_TYPE_BUFFER_END`, it means you have reached the last event in the buffer.

Return Value

This function returns a pointer to a `LumberjackBuffer` structure that Lumberjack has filled up. `NULL` is returned if there are currently no filled buffers.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

<:kernel:lumberjack.h>, libc.a

See Also

`ReleaseLumberjackBuffer()`, `DumpLumberjackBuffer()`, `CreateLumberjack()`, `DeleteLumberjack()`, `LogEvent()`, `ControlLumberjack()`

ReleaseLumberjackBuffer

Return a logging buffer to the Lumberjack system so it can be reused.

Synopsis

```
void ReleaseLumberjackBuffer(LumberjackBuffer *lb);
```

Description

Lumberjack, the Portfolio logging service, maintains a list of buffers into which it logs events during system execution. You can obtain pointers to full log buffers in order to display their contents using the `ObtainLumberjackBuffer()` folio call. Once you are done processing the contents of a buffer, you call `ReleaseLumberjackBuffer()` to return the buffer to Lumberjack so it can be reused.

Arguments

lb

The log buffer to return to Lumberjack, as obtained from `ObtainLumberjackBuffer()`. May be NULL in which case this function does nothing.

Implementation

Folio call implemented in Kernel folio V27.

Notes

... and I'm ok.

Associated Files

<:kernel:lumberjack.h>, libc.a

See Also

`ObtainLumberjackBuffer()`, `DumpLumberjackBuffer()`, `CreateLumberjack()`, `DeleteLumberjack()`, `LogEvent()`, `ControlLumberjack()`

AllocMem

Allocates a block of memory.

Synopsis

```
void *AllocMem(int32 memSize, uint32 memFlags);
```

Description

This function allocates a block of memory for use by the current task. You use `FreeMem()` to free a block of memory that was allocated with `AllocMem()`.

If there is insufficient memory in a task's free memory list to allocate the needed block, the kernel automatically attempts to obtain more memory pages from the system page pool. These pages become the property of the task and get added to its free memory list. They remain the property of the task until they are returned to the system page pool by calling `ScavengeMem()`.

When a task dies, any pages of memory it owns automatically get returned to the system page pool.

Memory allocated by a thread is owned by the parent task, and not by the thread itself. Therefore, when a thread dies, memory it allocated remains allocated until it is explicitly freed by the parent task (or another thread in the same task family), or if the parent task dies.

Arguments

`memSize`

The size of the memory block to allocate, in bytes.

`memFlags`

Flags that specify some options for this memory allocation.

Flags

These are the possible values for the `memFlags` argument.

`MEMTYPE_NORMAL`

Allocate standard memory. This flag must currently always be supplied when allocating memory.

The following flags specify some options concerning the allocation:

`MEMTYPE_FILL`

Sets every byte in the memory block to the value of the lower eight bits of the flags argument. If this bit is not set, the previous contents of the memory block are not changed.

`MEMTYPE_TRACKSIZE`

Tells the kernel to track the size of this memory allocation. When you allocate memory with this flag, you must specify `TRACKED_SIZE` as the size when freeing the memory using `FreeMem()`. This flag saves you the work of tracking the allocation's size. Keep in mind that it slightly increases the memory overhead of the allocation.

Return Value

Returns a pointer to the memory block that was allocated or `NULL` if there was not enough memory available.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:mem.h>, libc.a

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command line. Refer to the CreateMemDebug() function for more details.

See Also

AllocMemMasked(), FreeMem(), GetMemTrackSize()

AllocMemAligned

Allocates a block of memory aligned to a particular boundary.

Synopsis

```
void *AllocMemAligned(int32 memSize, uint32 memFlags,  
                      uint32 alignment);
```

Description

This macro allocates a block of memory for use by the current task. You use `FreeMem()` to free a block of memory that was allocated with `AllocMemAligned()`.

The difference between this macro and `AllocMem()` is that you can specify a particular alignment for the memory block. For example, you can allocate a block of memory which is aligned on a 4K boundary. This capability is very useful when interfacing to many kinds of hardware devices which have particular alignment needs. You can also use the `ALLOC_ROUND()` macro in `<:kernel:mem.h>` in order to round up the allocation size to a particular multiple.

If there is insufficient memory in a task's free memory list to allocate the needed block, the kernel automatically attempts to obtain more memory pages from the system page pool. These pages become the property of the task and get added to its free memory list. They remain the property of the task until they are returned to the system page pool by calling `ScavengeMem()`.

When a task dies, any pages of memory it owns automatically get returned to the system page pool.

Memory allocated by a thread is owned by the parent task, and not by the thread itself. Therefore, when a thread dies, memory it allocated remains allocated until it is explicitly freed by the parent task (or another thread in the same task family), or if the parent task dies.

Arguments

`memSize`

The size of the memory block to allocate, in bytes.

`memFlags`

Flags that specify some options for this memory allocation.

`alignment`

The alignment requirement. The address of the returned memory block will be aligned on a multiple of this value. This parameter must be a power of 2 and at be larger or equal to 8.

Flags

These are the possible values for the `memFlags` argument.

`MEMTYPE_NORMAL`

Allocate standard memory. This flag must currently always be supplied when allocating memory.

The following flags specify some options concerning the allocation:

`MEMTYPE_FILL`

Sets every byte in the memory block to the value of the lower eight bits of the flags argument. If this bit is not set, the previous contents of the memory block are not changed.

MEMTYPE_TRACKSIZE

Tells the kernel to track the size of this memory allocation. When you allocate memory with this flag, you must specify `TRACKED_SIZE` as the size when freeing the memory using `FreeMem()`. This flag saves you the work of tracking the allocation's size. Keep in mind that it slightly increases the memory overhead of the allocation.

Return Value

Returns a pointer to the memory block that was allocated or `NULL` if there was not enough memory available.

Implementation

Macro implemented in `<:kernel:mem.h>` V27.

Associated Files

`<:kernel:mem.h>`, `libc.a`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMem()`, `FreeMem()`, `GetMemTrackSize()`

AllocMemMasked

Allocates a block of memory with specific bits set or cleared in its address.

Synopsis

```
void *AllocMemMasked(int32 memSize, uint32 memFlags,  
                    uint32 careBits, uint32 stateBits);
```

Description

This function allocates a block of memory for use by the current task. You use `FreeMem()` to free a block of memory that was allocated with `AllocMemMasked()`.

The difference between this function and `AllocMem()` is that you can specify that certain bits be set or cleared in the address of returned memory block. This function is useful to allocate memory aligned to a particular boundary in memory. See the `AllocMemAligned()` macro for more details on this.

Although this function has a deceptively generic appearance, it is in fact meant to support the M2 triangle engine. For best performance, the TE wants the Z buffer allocated on alternate 4K pages from the main frame buffer. That is, if the Z buffer starts on an even numbered 4K page, the frame buffer should start on an odd numbered 4K page.

Using this function, you can allocate your frame buffer, and then by passing the right bit masks to this function, you can allocate the Z buffer on an alternate 4K boundary.

If there is insufficient memory in a task's free memory list to allocate the needed block, the kernel automatically attempts to obtain more memory pages from the system page pool. These pages become the property of the task and get added to its free memory list. They remain the property of the task until they are returned to the system page pool by calling `ScavengeMem()`.

When a task dies, any pages of memory it owns automatically get returned to the system page pool.

Memory allocated by a thread is owned by the parent task, and not by the thread itself. Therefore, when a thread dies, memory it allocated remains allocated until it is explicitly freed by the parent task (or another thread in the same task family), or if the parent task dies.

Arguments

memSize

The size of the memory block to allocate, in bytes.

memFlags

Flags that specify some options for this memory allocation.

careBits

Indicates which bits of the returned address matter to you. Bits set to 1 mean that you care about those bits, while bits set to 0 indicate you don't care what the value of these bits are.

stateBits

For those bits you care about, this indicates what their state should be. So for example, if you want the returned pointer to be allocated on a 4K boundary, it means you want the pointer returned to have the bottom 12 bits clear. You therefore would pass `careBits` of `0x00000fff` and `stateBits` of 0.

Flags

These are the possible values for the memFlags argument.

MEMTYPE_NORMAL

Allocate standard memory. This flag must currently always be supplied when allocating memory.

The following flags specify some options concerning the allocation:

MEMTYPE_FILL

Sets every byte in the memory block to the value of the lower eight bits of the flags argument. If this bit is not set, the previous contents of the memory block are not changed.

MEMTYPE_TRACKSIZE

Tells the kernel to track the size of this memory allocation. When you allocate memory with this flag, you must specify `TRACKED_SIZE` as the size when freeing the memory using `FreeMem()`. This flag saves you the work of tracking the allocation's size. Keep in mind that it slightly increases the memory overhead of the allocation.

Return Value

Returns a pointer to the memory block that was allocated or `NULL` if there was not enough memory available.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:mem.h>`, `libc.a`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMem()`, `FreeMem()`, `GetMemTrackSize()`

AllocMemPages

Allocates whole pages of memory.

Synopsis

```
void *AllocMemPages(int32 memSize, uint32 memFlags);
```

Description

This function allocates pages of memory directly from the system's free page pool, bypassing the current task's free memory list. This call is seldom used by client code, and is meant mostly as a support function to higher-level memory managers.

When a task dies, any pages of memory it owns automatically get returned to the system page pool.

Memory allocated by a thread is owned by the parent task, and not by the thread itself. Therefore, when a thread dies, memory it allocated remains allocated until it is explicitly freed by the parent task (or another thread in the same task family), or if the parent task dies.

Arguments

memSize

The size of the memory block to allocate, in bytes.

memFlags

Flags that specify some options for this memory allocation.

Flags

These are the possible values for the memFlags argument.

MEMTYPE_NORMAL

Allocate standard memory. This flag must currently always be supplied when allocating memory.

The following flags specify some options concerning the allocation:

MEMTYPE_FILL

Sets every byte in the memory block to the value of the lower eight bits of the flags argument. If this bit is not set, the previous contents of the memory block are not changed.

Return Value

Returns a pointer to the memory block that was allocated or NULL if there was not enough memory available.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:mem.h>, libc.a

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command line. Refer to the CreateMemDebug() function for more details.

See Also

`FreeMemPages()`, `ControlMem()`

AllocMemTrack

Allocates a block of memory with size-tracking.

Synopsis

```
void *AllocMemTrack(int32 memSize);
```

Description

This convenience macro allocates a block of memory using `AllocMem()`, requesting size-tracking (that is, using the `MEMTYPE_TRACKSIZE` flag).

You use `FreeMemTrack()` to free a block of memory that was allocated by `AllocMemTrack()`.

Size-tracking saves you the work of tracking the allocation size. Keep in mind that it slightly increases the memory overhead of the allocation.

Arguments

`memSize`

The size of the memory block to allocate, in bytes.

Return Value

Returns a pointer to the memory block that was allocated or `NULL` if there was not enough memory available.

Implementation

Macro implemented in `<:kernel:mem.h>` V27.

Associated Files

`<:kernel:mem.h>`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMemTrackWithOptions()`, `FreeMemTrack()`, `AllocMem()`,
`AllocMemMasked()`, `GetMemTrackSize()`

AllocMemTrackWithOptions

Allocates a block of memory with size-tracking and other options.

Synopsis

```
void *AllocMemTrackWithOptions(int32 memSize, uint32 memFlags);
```

Description

This convenience macro allocates a block of memory using `AllocMem()`, requesting size-tracking (that is, `MEMTYPE_TRACKSIZE`) and allowing you to supply other `memFlags` (e.g. `MEMTYPE_FILL`).

You use `FreeMemTrack()` to free a block of memory that was allocated with `AllocMemTrackWithOptions()`.

Size-tracking saves you the work of tracking the allocation's size. Keep in mind that it slightly increases the memory overhead of the allocation.

Arguments

`memSize`
The size of the memory block to allocate, in bytes.

`memFlags`
Flags that specify some options for this memory allocation.

Flags

The following flags specify some options concerning the allocation:

`MEMTYPE_FILL`
Sets every byte in the memory block to the value of the lower eight bits of the flags argument. If this bit is not set, the previous contents of the memory block are not changed.

Return Value

Returns a pointer to the memory block that was allocated or `NULL` if there was not enough memory available.

Implementation

Macro implemented in `<:kernel:mem.h>` V27.

Associated Files

`<:kernel:mem.h>`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMemTrack()`, `FreeMemTrack()`, `AllocMem()`, `AllocMemMasked()`,
`GetMemTrackSize()`

ControlMem

Controls memory permissions and ownership.

Synopsis

```
Err ControlMem(void *mem, int32 memSize, int32 cmd, Item task);
```

Description

When a task allocates memory, it becomes the owner of that memory. Other tasks cannot write to the memory unless they are given permission by its owner. A task can give another task permission to write to one or more of its memory pages, revoke write permission that was previously granted, or transfer ownership of memory to another task or the system by calling `ControlMem()`. Using `ControlMem()` a task can also make a memory range persistent or retain its contents across system reboot.

Each page of memory has a control status that specifies which task owns the memory, which tasks can write to it, and if the memory is persistent. Calls to `ControlMem()` change the control status for entire pages. If the `p` and `size` arguments (which specify the memory to change) specify any part of a page, the changes apply to the entire page.

A task can grant write permission for pages that it owns to any number of tasks. To accomplish this, the task must make a separate call to this function for each task that is to be granted write permission.

A task that calls `ControlMem()` must own the memory whose control status it is changing, with one exception: A task that has write access to memory it doesn't own can relinquish its write access by using `MEMC_NOWRITE` as the value of the `cmd` argument. If a task transfers ownership of memory, it still retains write access.

A task can use `ControlMem()` to prevent itself from writing to memory it owns. This may be useful during debugging to prevent one section of your code from stomping on the data for another section of the code.

A task can use `ControlMem()` to return ownership of memory pages to the system, thereby returning them to the system page pool. You can do this by using 0 as the value of the `task` argument.

A task can use `ControlMem()` to request memory pages to be marked persistent across reboot of an application. When the pages are marked persistent, they are returned to the system but the caller retains his current write privilege on those pages. After a system reset, if the same application that marked memory pages persistent (before the reboot) is run, the persistent memory pages are not allocated to any task. The value of the `task` argument must be 0 for this call. Currently, only one memory range can be marked as persistent.

Arguments

`mem`

A pointer to the memory whose control status to change.

`memSize`

The amount of memory for which to change the control status, in bytes. If the `memSize` and `mem` arguments specify any part of a page, the control status is changed for the entire page.

`cmd`

A constant that specifies the change to be made to the control status; possible values are listed below.

task

The item number of the task for which to change the control status or 0 for global changes.

The possible values of "cmd" are:

MEMC_OKWRITE

Grants permission to write to this memory to the task specified by the task argument, or to all tasks if the task argument is 0. When granting write permission to all tasks, it essentially makes the memory completely unprotected. To undo the effect of doing this, you must call `ControlMem()` with `MEMC_NOWRITE` and a task of 0, which removes write access for all tasks, and then call `ControlMem()` with `MEMC_OKWRITE` and the current task's item to allow the current task to write to the memory.

MEMC_NOWRITE

Revokes permission to write to this memory from the task specified by the task argument. If task is 0, revokes write permission for all tasks including the current task.

MEMC_GIVE

If the calling task is the owner of the memory, this transfers ownership of the memory to the task specified by the task argument. If task is 0, it gives the memory back to the system page pool.

MEMC_PERSISTENT

If the calling task is the owner of the memory, this transfers ownership of the memory to the system and marks the memory as persistent across reboot of the application.

Return Value

Returns ≥ 0 if the change was successful or a negative error code for failure. Possible error codes currently include:

BADITEM

The task argument does not specify a current task or is not 0 for `MEMC_PERSISTENT` cmd.

ER_Kr_BadMemCmd

The cmd argument is not one of the valid values.

ER_BadPtr

The mem argument is not a valid pointer to memory.

NOSUPPORT

The current task is attempting to set more than one persistent memory range.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:mem.h>`, `libc.a`

See Also

`ScavengeMem()`

ControlMemDebug

Controls what MemDebug does and doesn't do.

Synopsis

```
Err ControlMemDebug(uint32 controlFlags);
```

Description

This function lets you control various options that determine what MemDebug does.

Arguments

controlFlags

A set of bit flags controlling various MemDebug options. See below.

Flags

The control flags can be any of:

MEMDEBUGF_ALLOC_PATTERNS

When this flag is set, it instructs MemDebug to fill newly allocated memory with the constant MEMDEBUG_ALLOC_PATTERN. Doing so will likely cause your program to fail in some way if it tries to read newly allocated memory without first initializing it. Note that this option has no effect if memory is allocated using the MEMTYPE_FILL memory flag.

MEMDEBUGF_FREE_PATTERNS

When this flag is set, it instructs MemDebug to fill memory that is being freed with the constant MEMDEBUG_FREE_PATTERN. Doing so will likely cause your program to fail in some way if it tries to read memory that has been freed.

MEMDEBUGF_PAD_COOKIES

When this flag is set, it causes MemDebug to put special memory cookies in the 16 bytes before and 16 bytes after every block of memory allocated. When a memory block is freed, the cookies are checked to make sure that they have not been altered. This option makes sure that your program is not writing outside the bounds of memory it allocates. This option requires an extra overhead of 32 bytes per allocation.

MEMDEBUGF_BREAKPOINT_ON_ERRORS

When this flag is set, MemDebug automatically invokes the debugger if an error is detected. Errors include such things as mangled pad cookies, incorrect size for a FreeMem() call, etc. Normally, MemDebug simply prints out the error to the debugging terminal and keeps executing.

MEMDEBUGF_CHECK_ALLOC_FAILURES

When this flag is set, MemDebug emits a message when a memory allocation call fails due to lack of memory. This is useful to track down where in a program memory is running out.

MEMDEBUGF_KEEP_TASK_DATA

MemDebug maintains some task-specific statistics about memory allocations performed by that task. This information gets displayed by DumpMemDebug(). Whenever all of the memory allocated by a thread is freed, or when a task dies, the data structure holding the statistics for that task automatically gets freed by MemDebug. This is undesirable if you wish to dump out statistics of the code just before a program exits. Setting this flag causes the data structure not to be freed, making the statistical information available to DumpMemDebug().

Return Value

Returns ≥ 0 if successful or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:kernel:mem.h>`, `libc.a`

See Also

`CreateMemDebug()`,
`SanityCheckMemDebug()`

`DeleteMemDebug()`,

`DumpMemDebug()`,

CreateMemDebug

Initializes MemDebug, the Portfolio memory debugging package.

Synopsis

```
Err CreateMemDebug(const TagArg *tags);
```

Description

This function creates the needed data structures and initializes them as needed for MemDebug, the system memory debugging package.

MemDebug provides a general-purpose mechanism to track and validate all memory allocations done in the system. Using MemDebug, you can easily determine where memory leaks occur within a program, and find illegal uses of the memory subsystem.

To enable memory debugging in a program, do the following:

- * Add a call to `CreateMemDebug()` as the first statement in the `main()` routine of your program.
- * Add calls to `DumpMemDebug()` and `DeleteMemDebug()` as the last statements in the `main()` routine of your program.
- * Recompile your entire project with `MEMDEBUG` defined on the compiler's command-line (done by using `-DMEMDEBUG`)

With these steps taken, all memory allocations done by your program will be tracked, and specially managed. On exiting your program, any memory left allocated will be displayed to the debugging terminal, along with the line number and source file where the memory was allocated from.

In addition, MemDebug makes sure that illegal or dangerous uses of memory are detected and flagged. Most messages generated by the package indicate the offending source file and line within your source code where the problem originated.

When all options are turned on, MemDebug will check and report the following problems:

- * memory allocations with a size ≤ 0
- * memory free with a bogus memory pointer
- * memory free with a size not matching the size used when the memory was allocated
- * cookies on either side of all memory allocations are checked to make sure they are not altered from the time a memory allocation is made to the time the memory is released. This would indicate that something is writing beyond the bounds of allocated memory.

When source code is recompiled with `MEMDEBUG` defined, then all memory allocation and deallocation calls are vectored through special versions of these routines which track the source file and line number where the calls are made from. If memory is allocated from code not recompiled with `MEMDEBUG` (say if a folio allocates memory on your behalf), then the source file and line information is not available.

By calling the `DumpMemDebug()` function at any time within your program, you can get a detailed listing of all memory currently allocated, showing from which source line and source file the allocation occurred.

Arguments

tags

This is reserved for future use and should currently always be NULL.

Return Value

Returns ≥ 0 if successful or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:kernel:mem.h>`, `libc.a`

Notes

You should make sure to turn off memory debugging prior to creating the final version of your program. Enabling memory debugging incurs an overhead of currently 32 bytes per allocation made. If you use the `MEMDEBUGF_PAD_COOKIES` option, this overhead grows to 64 bytes per allocation.

In addition, specifying the `MEMDEBUGF_ALLOC_PATTERNS` and `MEMDEBUGF_FREE_PATTERNS` options will slow down memory allocations and free operations, due to the extra work of filling the memory with the patterns.

When reporting errors to the debugging terminal, the memory debugging subsystem will normally print the source file and line number where the error occurred. When using link libraries which have not been recompiled with `MEMDEBUG` defined, the memory debugging subsystem will still be able to track the allocations, but will not report the source file or line number where the error occurred. It will report `< unknown source file >` instead.

See Also

`ControlMemDebug()`, `DeleteMemDebug()`, `DumpMemDebug()`,
`SanityCheckMemDebug()`

DeleteMemDebug

Releases memory debugging resources.

Synopsis

```
Err DeleteMemDebug(void);
```

Description

Deletes any resources allocated by `CreateMemDebug()` and any resources to perform memory debugging.

This call is generally very risky to make if the `MEMDEBUGF_PAD_COOKIES` option was being used. In such a case, it is a good idea to reboot the system once the test is done running. When testing a program, you can also simply exit your program without turning off memory debugging. This will leave memory debugging active, and will avoid any problems associated with left over pad cookies,

Return Value

Returns ≥ 0 if successful, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:kernel:mem.h>`, `libc.a`

See Also

`CreateMemDebug()`,
`SanityCheckMemDebug()`

`ControlMemDebug()`,

`DumpMemDebug()`,

DumpMemDebug

Dumps memory allocation debugging information.

Synopsis

```
Err DumpMemDebug(const TagArg *tags);
```

```
Err DumpMemDebugVA(uint32 tag, ...);
```

Description

This function outputs a table showing all memory currently allocated through the memory debugging code. This table shows the allocation size, address, as well as the source file and the source line where the allocation took place.

This function also outputs statistics about general memory allocation patterns. This includes the number of memory allocation calls that have been performed, the maximum number of bytes allocated at any one time, current amount of allocated memory, etc. All this information is displayed on a per-thread basis, as well as globally for all threads.

To use this function, the memory debugging code must have been previously initialized using `CreateMemDebug()`.

Arguments

`tags`

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with `TAG_END`.

`DUMPMEMDEBUG_TAG_TASK (Item)`

Controls which task's information should be displayed. You can pass the item number of any task to display. Passing 0 prints the task information of all tasks. If this tag is not supplied, the default is to only display the information for the current task.

`DUMPMEMDEBUG_TAG_SUPER (bool)`

When set to `TRUE`, causes information to be displayed about allocations done in supervisor mode by the system.

Return Value

Returns ≥ 0 if successful, or a negative error code if not. Current

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

`<:kernel:mem.h>`, `libc.a`

See Also

`CreateMemDebug()`, `ControlMemDebug()`, `DeleteMemDebug()`,
`SanityCheckMemDebug()`

FreeMem

Frees memory that was allocated with `AllocMem()` or `AllocMemAligned()`.

Synopsis

```
void FreeMem(void *mem, int32 memSize);
```

Description

This function frees memory that was previously allocated by a call to `AllocMem()`. The size argument specifies the number of bytes to free.

The memory is added to the list of available memory for the current task. The pages of RAM containing the block of memory being freed are not returned to the system page pool and remain the property of the current task. You must call `ScavengeMem()` in order to return these pages.

Arguments

`mem`

The memory block to free. This value may be `NULL`, in which case this function just returns.

`memSize`

The size of the block to free, in bytes. This must be the same size that was specified when the block was allocated. If the memory being freed was allocated using `MEMTYPE_TRACKSIZE`, this argument should be set to `TRACKED_SIZE`.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:mem.h>`, `libc.a`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMem()`, `ScavengeMem()`

FreeMemPages

Frees memory that was allocated with `AllocMemPages()`.

Synopsis

```
void FreeMemPages(void *mem, int32 memSize);
```

Description

This function frees memory that was previously allocated by a call to `AllocMemPages()`. The size argument specifies the number of bytes to free.

The memory is immediately returned to the system's free page pool, where it becomes available for reallocation by another task or by the system.

Arguments

`mem`

The memory block to free. This value may be NULL, in which case this function just returns.

`memSize`

The size of the block to free, in bytes. This must be the same size that was specified when the block was allocated.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:mem.h>, libc.a

Notes

You can enable memory debugging in your application by compiling your entire project with the MEMDEBUG value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMemPages()`

FreeMemTrack

Frees memory that was allocated with
`AllocMemTrack()` or
`AllocMemTrackWithOptions()`.

Synopsis

```
void FreeMemTrack(void *mem);
```

Description

This convenience macro frees memory (via `FreeMem()`) that was previously allocated by a call to `AllocMemTrack()` or `AllocMemTrackWithOptions()`. The system is tracking this memory block's size for your convenience, so `FreeMemTrack()` doesn't need a `memSize` argument.

The memory is added to the list of available memory for the current task. The pages of RAM containing the block of memory being freed are not returned to the system page pool and remain the property of the current task. You must call `ScavengeMem()` in order to return these pages.

Arguments

`mem`

The memory block to free. This value may be `NULL`, in which case this function just returns.

Implementation

Macro implemented in `<:kernel:mem.h>` V27.

Associated Files

`<:kernel:mem.h>`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMemTrack()`, `AllocMemTrackWithOptions()`, `AllocMem()`, `FreeMem()`,
`ScavengeMem()`

GetMemInfo

Gets information about available memory.

Synopsis

```
void GetMemInfo(MemInfo *minfo, uint32 infoSize, uint32 memFlags);
```

Description

This function returns information about the amount of memory that is currently available. The information about available memory is returned in a MemInfo structure.

The MemInfo structure contains the following fields:

`minfo_TaskAllocatedPages`

Specifies the number of pages the current task owns.

`minfo_TaskAllocatedBytes`

Specifies the number of bytes allocated by the current task.

`minfo_FreePages`

Specifies the number of unallocated pages in the system.

`minfo_LargestFreePageSpan`

Specifies the largest number of contiguous free pages in the system.

`minfo_SystemAllocatedPages`

Specifies the number of pages allocated by the system to store items and other system-private allocations.

`minfo_SystemAllocatedBytes`

Specifies the number of bytes allocated by the system to store items and other system-private allocations.

`minfo_OtherAllocatedPages`

Specifies the number of pages allocated by tasks other than the current task.

Arguments

`minfo`

A pointer to a MemInfo structure where the information will be stored.

`infoSize`

The size in bytes of the MemInfo structure.

`memFlags`

This must currently always be MEMTYPE_NORMAL.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:mem.h>, libc.a

Caveats

The information returned by `GetMemInfo()` is inherently flawed, since you are existing in a multitasking

environment. Memory can be allocated or freed asynchronous to the operation of the task calling `GetMemInfo()`.

See Also

`AllocMem()`, `FreeMem()`, `ScavengeMem()`

GetMemTrackSize

Gets the size of a block of memory allocated with MEMTYPE_TRACKSIZE.

Synopsis

```
int32 GetMemTrackSize(const void *mem);
```

Description

This function returns the size that was used to allocate a block of memory. The block of memory must have been allocated using the MEMTYPE_TRACKSIZE flag, otherwise this function will return garbage.

Arguments

mem

Pointer obtained from AllocMem(). The block of memory must have been allocated using the MEMTYPE_TRACKSIZE flag, otherwise the value returned by this function will be random.

Return Value

Returns the size in bytes of the memory block. This size corresponds to the size provided to AllocMem() when the block was first allocated.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:mem.h>, libc.a

See Also

AllocMem(), FreeMem()

GetPageSize

Gets the size in bytes of a page of memory.

Synopsis

```
int32 GetPageSize(uint32 memFlags);
```

Description

This function gets the number of bytes in a page of memory.

Arguments

memFlags

The type of memory to inquire about. This value must currently always be MEMTYPE_NORMAL.

Return Value

Returns the number of bytes in a page of memory, or a negative error code if some illegal flags were specified.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:mem.h>, libc.a

IsMemOwned

Determines whether a region of memory is owned by the current task.

Synopsis

```
bool IsMemOwned(const void *mem, int32 memSize);
```

Description

This function considers the described block of the address space in relation to the pages of memory the current task owns. This function returns TRUE If the current task owns all pages in the specified memory block, and FALSE if it doesn't.

Arguments

mem
A pointer to the start of the block.

memSize
The number of bytes in the block.

Return Value

Returns TRUE if the block is entirely owned by the calling task, and FALSE if it cannot.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:mem.h>, libc.a

See Also

IsMemReadable(), IsMemWritable()

IsMemReadable

Determines whether a region of memory is fully readable by the current task.

Synopsis

```
bool IsMemReadable(const void *mem, int32 memSize);
```

Description

This function considers the described block of the address space in relation to the known locations of RAM in the system, and returns TRUE if the block is entirely contained within RAM accessible by the current task, and FALSE otherwise.

Arguments

mem

A pointer to the start of the block.

memSize

The number of bytes in the block.

Return Value

Returns TRUE if the block is entirely readable by the current task, or FALSE if any part of the block is not.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:mem.h>, libc.a

See Also

IsMemWritable(), IsMemOwned()

IsMemWritable

Determines whether a region of memory is fully writable by the current task.

Synopsis

```
bool IsMemWritable(const void *mem, int32 memSize);
```

Description

This function considers the described block of the address space in relation to the pages of memory the current task has write access to. This function returns TRUE if the current task can write to the memory block, and FALSE if it cannot.

Arguments

mem
A pointer to the start of the block.

memSize
The number of bytes in the block.

Return Value

Returns TRUE if the block can be written to by the calling task, and FALSE if it cannot.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:mem.h>, libc.a

See Also

IsMemReadable(), IsMemOwned()

RationMemDebug

Rations memory allocations to test failure paths.

Synopsis

```
Err RationMemDebug(const TagArg *tags);
```

```
Err RationMemDebugVA(uint32 tag, ...);
```

Description

This function lets you cause selected memory allocations to fail, allowing various failure paths to be tested.

The many tags supported let you tailor when and under which conditions memory allocations are to fail.

Arguments

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

RATIONMEMDEBUG_TAG_ACTIVE (bool)

This tag lets you control whether memory rationing is turned on. Passing TRUE turns rationing on, while passing FALSE turns it off.

RATIONMEMDEBUG_TAG_TASK (Item)

Specifies that rationing is to only occur for a specific task. You supply the item number of the task to this call. If you pass a task item of 0, it means that all tasks should be rationed.

RATIONMEMDEBUG_TAG_MINSIZE (uint32)

Lets you specify the minimum size of allocations to ration. Allocations smaller than this size will never be rationed.

RATIONMEMDEBUG_TAG_MAXSIZE (uint32)

Lets you specify the maximum size of allocations to ration. Allocations larger than this size will never be rationed.

RATIONMEMDEBUG_TAG_COUNTDOWN (uint32)

Specifies a countdown of allocations before rationing should start. Rationing will be disabled until that many allocations are performed.

RATIONMEMDEBUG_TAG_INTERVAL (uint32)

Specifies the period over which the rationing occurs. Only a single allocation per interval is rationed.

RATIONMEMDEBUG_TAG_RANDOM (bool)

Specifies that within the rationing interval, a random allocation should fail. If random mode is turned off, the sequencing of rationed allocations will be consistent.

RATIONMEMDEBUG_TAG_VERBOSE (bool)

When set to TRUE, specifies that when a rationing occurs, information should be printed about the allocation that was denied.

RATIONMEMDEBUG_TAG_BREAKPOINT_ON_RATIONING (bool)

When set to TRUE, specifies that a debugger breakpoint should be triggered whenever rationing occurs.

RATIONMEMDEBUG_TAG_SUPER (bool)

When set to TRUE, specifies that supervisor allocations should be rationed just like normal user allocations. This will make things like item allocations start to fail.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in kernel folio V30.

Associated Files

<:kernel:mem.h>

See Also

`ControlMemDebug()`, `CreateMemDebug()`, `DeleteMemDebug()`

ReallocMem

Reallocates a block of memory to a different size.

Synopsis

```
void *ReallocMem(void *mem, int32 oldSize, int32 newSize,  
                uint32 memFlags);
```

Description

This function reallocates a block of memory to a different size.

When making a block smaller, the excess memory at the end of the block is returned to the task's free memory list, where it becomes available for future allocations.

When making a block larger, an attempt is made to expand the memory block in place. If this is not possible because the region following the memory block has already been allocated, then an attempt will be made to move the memory block to a different sufficiently large area in memory. In such a case, this function will run much slower since it will need to copy the contents of the old memory area into the new area.

NOTE: Making a block larger is not currently supported.

Arguments

mem

The memory block to reallocate. This value may be NULL, in which case this function just returns NULL.

oldSize

The old size of the memory block, in bytes. This must be the same size that was specified when the block was allocated. If the memory block was allocated using MEMTYPE_TRACKSIZE, this argument should be set to TRACKED_SIZE.

newSize

The new size of the block of memory.

memFlags

Flags that specify some options for this memory allocation.

Flags

These are the possible values for the memFlags argument.

MEMTYPE_NORMAL

Allocate standard memory. This flag must currently always be supplied when reallocating memory.

The following flags specify some options concerning the allocation:

MEMTYPE_FILL

Sets every new byte in the memory block to the value of the lower eight bits of the flags argument. If this bit is not set, the previous contents of the new extent of the memory block are not changed. This flag has no effect when making a memory block smaller.

MEMTYPE_TRACKSIZE

Tells the kernel to track the size of this memory allocation. When you allocate memory with

this flag, you must specify `TRACKED_SIZE` as the size when freeing the memory using `FreeMem()`. This flag saves you the work of tracking the allocation's size. Keep in mind that it slightly increases the memory overhead of the allocation.

Return Value

Returns a pointer to the new memory block or `NULL` if there was not enough memory available. This pointer will often, but not always, be equal to the memory block pointer supplied to the function.

When this function returns `NULL`, the original block of memory is still allocated.

Implementation

Folio call implemented in Kernel folio V27.

Caveats

This function currently only allows a block of memory to shrink. Attempting to make a block bigger will always return `NULL`.

Associated Files

`<:kernel:mem.h>`, `libc.a`

Notes

You can enable memory debugging in your application by compiling your entire project with the `MEMDEBUG` value defined on the compiler's command line. Refer to the `CreateMemDebug()` function for more details.

See Also

`AllocMem()`, `FreeMem()`, `GetMemTrackSize()`

SanityCheckMemDebug

Checks all current memory allocations to make sure all the allocation cookies are intact

Synopsis

```
Err SanityCheckMemDebug(const char *banner, const TagArg *tags);
```

Description

This function checks all current memory allocations to see if any of the memory cookies have been corrupted. This is useful when trying to track down at which point in a program's execution memory cookies are being trashed.

Arguments

banner

Descriptive text to print before any status message displayed. May be NULL.

tags

This is reserved for future use and should currently always be NULL.

Return Value

Returns ≥ 0 if successful or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:mem.h>, libc.a

See Also

CreateMemDebug(), ControlMemDebug(), DeleteMemDebug(), DumpMemDebug()

ScavengeMem

Returns the current task's unused memory pages to the system page pool.

Synopsis

```
int32 ScavengeMem(void);
```

Description

This function finds pages of memory in the current task's free memory list that are currently unused and returns them to the system page pool, where they can be reallocated for other uses.

Return Value

Returns the amount of memory that was freed to the system page pool, in bytes.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:mem.h>, libc.a

See Also

AllocMem(), FreeMem()

CreateBufferedMsg

Creates a buffered message.

Synopsis

```
Item CreateBufferedMsg(const char *name, uint8 pri, Item msgPort,  
                      int32 dataSize);
```

Description

One of the ways tasks communicate is by sending messages to each other. This function creates an item for a buffered message (a message that includes an internal buffer for sending data to the receiving task).

The advantage of using a buffered message instead of a standard message is that the sending task doesn't need to keep the data block containing the message data after the message is sent. All the necessary data is included in the message.

The same message item can be resent any number of times. When you are finished with a message item, use `DeleteMsg()` to delete it.

You can use `FindNamedItem()` to find a message item by name.

Arguments

- name**
The optional name of the message.
- pri**
The priority of the message. This determines the position of the message in the receiving task's message queue and thus, how soon it is likely to be handled. A larger number specifies a higher priority.
- msgPort**
The item number of the message port at which to receive the reply, or 0 if no reply is expected.
- dataSize**
The maximum size of the message's internal buffer, in bytes.

Return Value

Returns the item number of the message or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsg()`, `CreateMsgPort()`, `CreateSmallMsg()`, `DeleteMsg()`,
`DeleteMsgPort()`, `SendMsg()`

CreateMsg

Creates a standard message.

Synopsis

```
Item CreateMsg(const char *name, uint8 pri, Item msgPort);
```

Description

One of the ways tasks communicate is by sending messages to other. This function creates a standard message. Standard messages require that data data to be communicated to the receiving task be contained in memory belonging to the sending task.

When you create a message, you can provide an optional reply port. Any task you send the message to will be able to reply it to you by calling `ReplyMsg()`. If you don't supply a reply port, then the act of sending the message to another task will also transfer the ownership of the message to the receiving task.

Arguments

name

Optional name for the message. Use NULL for no name.

pri

The priority of the message. This determines the position of the message in the receiving task's message queue and thus, how soon it is likely to be handled. A larger number specifies a higher priority.

msgPort

The item number of the message port at which to receive the reply, or 0 for no reply port.

Return Value

The item number of the message or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsgPort()`, `CreateBufferedMsg()`, `CreateSmallMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`

CreateMsgPort

Creates a message port.

Synopsis

```
Item CreateMsgPort(const char *name, uint8 pri, int32 sigMask);
```

Description

This function creates a message port item, which is used to receive messages from other tasks and threads.

You usually give a name to your message ports. Named ports can be used as a rendez-vous between tasks, since other tasks are able to find the message by name by using `FindMsgPort()`. The priority value you supply to this function is used to determine the sorting order of message ports in the list of ports maintained by the kernel. When you call `FindMsgPort()`, if there are multiple ports with the desired name, the one with the highest priority will be returned. See the `CreateUniqueMsgPort()` function for a way to create a message port with a name that is guaranteed to be unique.

When you no longer need a message port you use `DeleteMsgPort()` to delete it.

Arguments

name

Optional name of the message port, or NULL if the port should be unnamed. Unnamed ports cannot be found with `FindMsgPort()`.

pri

The priority of the message port. If you don't care, supply 0 for this argument.

sigMask

Whenever a message arrives at a message port, the kernel sends a signal to the owner of the message port. This argument lets you specify which signals the kernel should send to your task whenever a message arrives at the port being created. If you supply 0 for this argument, the kernel will allocate a signal bit automatically, and will free this bit when the message port is later deleted. You can look at the `mp_Signal` field of the `MsgPort` structure to determine which signals are sent when a message arrives at the port.

Return Value

The item number of the new message port or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

<:kernel:msgport.h>, `libc.a`

See Also

`CreateMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`, `SendSmallMsg()`

CreateSmallMsg

Creates a small message.

Synopsis

```
Item CreateSmallMsg(const char *name, uint8 pri, Item mp);
```

Description

This function creates a small message (a message that can contain up to eight bytes of data). Small messages are the fastest kind of messages you can send, as no data is copied, and no pointers need to be validated.

To create a standard message (a message in which any data to be communicated to the receiving task is contained in a data block allocated by the sending task), use `CreateMsg()`. To create a buffered message (a message that includes an internal buffer for sending data to the receiving task), use `CreateBufferedMsg()`.

The same message item can be resent any number of times. When you are finished with a message item, use `DeleteMsg()` to delete it.

You can use `FindNamedItem()` to find a message by name.

Arguments

name

The optional name of the message.

pri

The priority of the message. This determines the position of the message in the receiving task's message queue and thus, how soon it is likely to be handled. A larger number specifies a higher priority.

mp

The item number of the message port at which to receive the reply, or 0 if no reply is expected.

Return Value

Returns the item number of the message or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsg()`, `CreateMsgPort()`, `CreateBufferedMsg()`, `DeleteMsg()`,
`DeleteMsgPort()`, `SendMsg()`

CreateUniqueMsgPort

Creates a message port with a unique name.

Synopsis

```
Item CreateUniqueMsgPort(const char *name, uint8 pri, int32  
sigMask);
```

Description

This function works like `CreateMsgPort()`, except that it guarantees that no other message port item of the same name already exists, and that once this port created, no other port of the same name will be allowed to be created.

Refer to the `CreateMsgPort()` documentation for more information.

Arguments

name
The name of the message port.

pri
The priority of the message port.

sigMask
The signal to send when a message arrives at the port.

Return Value

The item number of the new message port or a negative error code for failure. If a port of the same name already existed when this call was made, the `ER_Kr_UniqueItemExists` error will be returned.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsg()`, `DeleteMsg()`, `DeleteMsgPort()`, `SendMsg()`, `SendSmallMsg()`

DeleteMsg

Deletes a message.

Synopsis

```
Err DeleteMsg(Item msg);
```

Description

Deletes the specified message and any resources that were allocated for it.

Arguments

`msg`
The item number of the message to be deleted.

Return Value

≥ 0 if successful or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:msgport.h>` V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsg()`

DeleteMsgPort

Deletes a message port.

Synopsis

```
Err DeleteMsgPort (Item msgPort)
```

Description

Deletes the specified message port. Any messages still left on the port are replied with a result code of BADITEM.

Arguments

msgPort
The item number of the message port to be deleted.

Return Value

>= 0 if successful or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:msgport.h>` V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsg()`, `CreateMsgPort()`

FindMsgPort

Finds a message port by name.

Synopsis

```
Item FindMsgPort(const char *name);
```

Description

This macro finds a message port with the specified name. The search is not case-sensitive.

Arguments

name
The name of the message port to find.

Return Value

Returns the item number of the message port that was found, or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:msgport.h>` V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`CreateMsgPort()`, `CreateUniqueMsgPort()`

GetMsg

Gets a message from a message port.

Synopsis

```
Item GetMsg(Item msgPort);
```

Description

This function gets the first message in the message queue for the specified message port and removes the message from the queue.

Once you have gotten a message, you can do a number of things. If the message is a reply to a message you sent out previously, you can either reuse the message, or delete it. If this a message being sent to you by another task, you can use `LookupItem()` to obtain a pointer to the Message structure and look at the various field therein to get more information.

If a task has gotten messages and proceeds to exit without replying them, the kernel will automatically reply the messages on behalf of the task. The messages will have a result code of `KILLED`.

Arguments

`msgPort`

The item number of the message port from which to get the message.

Return Value

The item number of the first message in the message queue or a negative error code for failure. Possible error codes currently include:

0

The queue is empty.

`BADITEM`

The mp argument does not specify a message port.

`NOTOWNER`

The message port specified by the mp argument is not owned by this task.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:msgport.h>, libc.a

See Also

`DeleteMsg()`, `GetThisMsg()`, `ReplyMsg()`, `ReplySmallMsg()`, `SendMsg()`,
`SendSmallMsg()`, `WaitPort()`

GetThisMsg

Gets a specific message.

Synopsis

```
Item GetThisMsg(Item msg);
```

Description

This function gets a specific message and removes it from the message port it is currently on.

A task that is receiving messages can use `GetThisMsg()` to get an incoming message. Unlike `GetMsg()` which gets the first message on a specific message port, `GetThisMsg()` can get any message from any of the task's message ports.

A task that has sent a message can use `GetThisMsg()` to get it back. If the receiving task hasn't already taken the message from its message port, `GetThisMsg()` removes it from the port. If the message is not on any port an error is returned.

In order to get a message, one or more of the following must be true:

- The current task owns the message.
- The current task owns the port the message is on.
- The current task is a member of the same task family as the owner of the message.
- The current task is a member of the same task family as the owner of the port the message is on.

If a task has gotten messages and proceeds to exit without replying them, the kernel will automatically reply the messages on behalf of the task. The messages will have a result code of `KILLED`.

Arguments

`msg`

The item number of the message to get.

Return Value

The item number of the message or a negative error code for failure. Possible error codes currently include:

`BADITEM`

The message argument does not specify a message.

`BADPRIV`

The calling task is not allowed to get this message.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

Caveats

Prior to V21, if the message was not on any message port this function still returned the item number of the message. In V21 and beyond, if the message is not on any port, the function returns an error.

See Also

CreateMsg(), GetMsg(), ReplyMsg(), ReplySmallMsg(), SendMsg(),
SendSmallMsg(), WaitPort()

ReplyMsg

Sends a reply to a message.

Synopsis

```
Err ReplyMsg(Item msg, int32 result,  
              const void *dataptr, int32 datasize);
```

Description

This function sends a reply to a standard or buffered message.

When a message is first created, a reply port may be provided by the creator. When you reply a message, it is like sending it to the reply port.

When replying a message, you supply a result code which typically indicates whether the operation triggered by the message was successful.

The meaning of the dataptr and datasize arguments depend on the type of message being replied. You can find out what type of message it is by looking at its msg.n_Flags field. If it is a small message, the value of the field is MESSAGE_SMALL. If it is a buffered message, the value of the field is MESSAGE_PASS_BY_VALUE. If neither bit is set, the message is a standard message.

If the message is a standard message, the dataptr and datasize arguments refer to a data block that your task allocates for returning reply data. For standard messages, the sending task and the replying task must each allocate their own memory blocks for message data. If the message is buffered, the data is copied into the internal buffer of the message. To reply a small message, use the ReplySmallMsg() function.

If a task has gotten messages and proceeds to exit without replying them, the kernel will automatically reply the messages on behalf of the task. The messages will have a result code of KILLED.

Arguments

msg

The message to reply, as gotten from GetMsg() or GetThisMsg().

result

A result code. This code is placed in the msg_Result field of the Message data structure before the reply is sent. The meaning of this code is undefined from the system's perspective; the code must be a value whose meaning is agreed upon by the calling and receiving tasks. In general, you should use negative values to specify errors and 0 to specify that no error occurred.

dataptr

See the "Description" section above for a description of this argument.

datasize

See the "Description" section above for a description of this argument.

Return Value

>= 0 if the reply was sent successfully or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`GetMsg()`, `GetThisMsg()`, `ReplySmallMsg()`, `SendMsg()`, `SendSmallMsg()`,
`WaitPort()`

ReplySmallMsg

Sends a reply to a small message.

Synopsis

```
Err ReplySmallMsg(Item msg, int32 result, uint32 val1, uint32 val2);
```

Description

This function sends a reply to a small message.

When a message is first created, a reply port may be provided by the creator. When you reply a message, it is like sending it to the reply port.

When replying a message, you supply a result code which typically indicates whether the operation triggered by the message was successful.

If a task has gotten messages and proceeds to exit without replying them, the kernel will automatically reply the messages on behalf of the task. The messages will have a result code of KILLED.

Arguments

msg

The message to reply, as gotten from `GetMsg()` or `GetThisMsg()`.

result

A result code. This code is placed in the `msg_Result` field of the Message data structure before the reply is sent. The meaning of this code is undefined from the system's perspective; the code must be a value whose meaning is agreed upon by the calling and receiving tasks. In general, you should use negative values to specify errors and 0 to specify that no error occurred.

val1

The first four bytes of data for the reply. This data is put into the `msg_DataPtr` field of the message structure.

val2

The last four bytes of data for the reply. This data is put into the `msg_DataSize` field of the message structure.

Return Value

≥ 0 if the reply was sent successfully or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`GetMsg()`, `GetThisMsg()`, `ReplyMsg()`, `SendMsg()`, `SendSmallMsg()`, `WaitPort()`

SendMsg

Sends a message.

Synopsis

```
Err SendMsg(Item msgPort, Item msg,  
            const void *dataptr, int32 datasize);
```

Description

This function sends a message to the specified message port. (To send a small message use `SendSmallMsg()`.)

The message is queued on the message port's list of messages according to its priority. Messages that have the same priority are queued in FIFO order.

Whether a message is standard or buffered is determined by how it was created: `CreateMsg()` creates a standard message, while `CreateBufferedMsg()` creates a buffered message.

A standard message is one whose message data belongs to the sending task. The receiving task just gets pointers to the data and reads from there, which means the data area for a standard message must remain valid until the receiving task is done reading from it.

A buffered message contains a data buffer within the message packet itself. When you send such a message, the data you supply is copied into the buffer. The receiving task then reads the data from the message's buffer, which means that you can immediately reuse your data buffer after having sent a buffered message since the receiving task will be reading the data from the message's buffer and not from your data buffer.

When sending a buffered message, `SendMsg()` checks the size of the data block to see whether it will fit in the message's buffer. If it won't fit, `SendMsg()` returns an error.

If a message is created without a reply port, then the act of sending the message also transfers the ownership of the message to the recipient task.

Arguments

`msgPort`

The item number of the message port to which to send the message.

`msg`

The item number of the message to send.

`dataptr`

A pointer to the message data, or NULL if there is no data to include in the message. For a standard message, the pointer specifies a data block owned by the sending task, which can later be read by the receiving task. For a buffered message, the pointer specifies a block of data to be copied into the message's internal data buffer.

`datasize`

The size of the message data, in bytes, or 0 if there is no data to include in the message.

Return Value

≥ 0 if the message was sent successfully or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:msgport.h>, libc.a

See Also

GetMsg(), GetThisMsg(), ReplyMsg(), ReplySmallMsg(), SendSmallMsg(),
WaitPort()

SendSmallMsg

Sends a small message.

Synopsis

```
Err SendSmallMsg(Item msgPort, Item msg, uint32 val1, uint32 val2);
```

Description

This function sends a small message. Small messages contain only 8 bytes of data. This function call is a synonym for `SendMsg()` and is provided merely as a convenience to avoid the need to cast the two data values to the types needed by `SendMsg()`. See the documentation for `SendMsg()` for more details.

Arguments

`msgPort`

The item number of the message port to which to send the message.

`msg`

The item number of the message to send.

`val1`

The first four bytes of message data. This data is put into the `msg_DataPtr` field of the message structure.

`val2`

The last four bytes of message data. This data is put into the `msg_DataSize` field of the message structure.

Return Value

≥ 0 if the message is sent successfully or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:msgport.h>`, `libc.a`

See Also

`GetMsg()`, `GetThisMsg()`, `ReplyMsg()`, `ReplySmallMsg()`, `SendMsg()`, `WaitPort()`

WaitPort

Waits for a message to arrive at a message port.

Synopsis

```
Item WaitPort(Item msgPort, Item msg);
```

Description

The function puts the calling task into wait state until a message arrives at the specified message port. When a task is in wait state, it uses no CPU time.

The task can wait for a specific message by providing the item number of the message as the msg argument. To wait for any incoming message, the task uses 0 as the value of the msg argument.

If the desired message is already in the message queue for the specified message port, the function returns immediately. If the message never arrives, the function may never return.

When the message arrives, the task is moved from the wait queue to the ready queue, the item number of the message is returned as the result, and the message is removed from the message port.

If a task has gotten messages and proceeds to exit without replying them, the kernel will automatically reply the messages on behalf of the task. The messages will have a result code of KILLED.

Arguments

msgPort

The item number of the message port to check for incoming messages.

msg

The item number of the message to wait for, or 0 if any message will do.

Return Value

The item number of the incoming message or a negative error code for failure.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

<:kernel:msgport.h>, libc.a

See Also

GetMsg(), ReplyMsg(), SendMsg()

ReadHardwareRandomNumber Gets a 32-bit random number.**Synopsis**

```
uint32 ReadHardwareRandomNumber(void);
```

Description

This function returns a random number generated by the hardware. This is useful for providing a seed to the software random-number generator.

Return Value

Returns an unsigned 32-bit random number. The number can be any integer from 0 to ($2^{32} - 1$), inclusive.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:random.h>, libc.a

See Also

`srand()`, `rand()`, `urand()`

ReadUniqueID

Gets the system unique id.

Synopsis

```
Err ReadUniqueID(UniqueID *id);
```

Description

3DO systems may be equipped with a unique ID chip that uniquely identifies each machine. This function lets you read the value of this ID chip.

The unique ID code is mainly useful to networked applications. It may also be used to determine whether a saved game is being used on the same machine it was originally created.

Arguments

id

Pointer to a UniqueID structure where the system's ID is stored.

Return Value

Returns ≥ 0 for success, or a negative error code for failure. Possible error codes currently include:

NOSUPPORT

The system doesn't have a unique ID code.

Implementation

Folio call implemented in Kernel folio V30.

Associated Files

<:kernel:uniqueid.h>

CreateSemaphore

Creates a semaphore.

Synopsis

```
Item CreateSemaphore( const char *name, uint8 pri )
```

Description

This function creates a semaphore item with the specified name and priority. You can use this function in place of `CreateItem()` to create the semaphore.

When you no longer need a semaphore, use `DeleteSemaphore()` to delete it.

If you give a name to the semaphore when creating it, other tasks will be able to locate the semaphore by using the `FindSemaphore()` function.

Arguments

name

The name of the semaphore, or NULL if it is unnamed.

pri

The priority of the semaphore; use 0 for now.

Return Value

Returns the item number of the semaphore or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V20`.

Associated Files

`<:kernel:semaphore.h>`, `libc.a`

See Also

`DeleteSemaphore()`, `LockSemaphore()`, `UnlockSemaphore()`

CreateUniqueSemaphore

Creates a semaphore with a unique name.

Synopsis

```
Item CreateUniqueSemaphore( const char *name, uint8 pri )
```

Description

This function creates a semaphore item with the specified name and priority. You can use this function in place of `CreateItem()` to create the semaphore.

When you no longer need a semaphore, use `DeleteSemaphore()` to delete it.

This function works much like `CreateSemaphore()`, except that it guarantees that no other semaphore item of the same name already exists. And once this semaphore created, no other semaphore of the same name is allowed to be created.

Arguments

name

The name of the semaphore.

pri

The priority of the semaphore; use 0 for now.

Return Value

Returns the item number of the semaphore or a negative error code for failure. If a semaphore of the same name already existed when this call was made, the `ER_Kr_UniqueItemExists` error will be returned.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:semaphore.h>`, `libc.a`

See Also

`DeleteSemaphore()`, `LockSemaphore()`, `UnlockSemaphore()`

DeleteSemaphore

Deletes a semaphore.

Synopsis

```
Err DeleteSemaphore( Item s )
```

Description

This macro deletes a semaphore that was created with `CreateSemaphore()`.

Arguments

`s`
The item number of the semaphore to be deleted.

Return Value

Returns ≥ 0 if successful or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:semaphore.h>` V20.

Associated Files

`<:kernel:semaphore.h>`, `libc.a`

See Also

`CreateSemaphore()`

FindSemaphore

Finds a semaphore by name.

Synopsis

```
Item FindSemaphore(const char *name);
```

Description

This macro finds a semaphore with the specified name. The search is not case-sensitive.

Arguments

name
The name of the semaphore to find.

Return Value

Returns the item number of the semaphore that was found, or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:semaphore.h>` V20.

Associated Files

`<:kernel:semaphore.h>`, `libc.a`

See Also

`CreateSemaphore()`, `CreateUniqueSemaphore()`

LockSemaphore

Locks a semaphore.

Synopsis

```
int32 LockSemaphore( Item s, uint32 flags )
```

Description

This function locks a semaphore.

Semaphores are used to control access to shared resources from different tasks or threads. By common agreement, tasks commit to locking a semaphore before accessing data that is associated with it.

Semaphores can be locked in one of two modes. In exclusive mode, only one task can have the semaphore locked at a time. In shared mode, any number of tasks can have the semaphore locked in shared mode. Typically, when you simply need to "read" a shared resource, you should lock the semaphore in shared mode. If you need to modify the resource, you should lock it in exclusive mode.

Semaphore locks nest. If you currently have a semaphore locked in exclusive mode and you try to relock it in shared mode, it is simply relocked in exclusive mode. It is not legal to relock a shared mode semaphore in exclusive mode.

Arguments

s

The item number of the semaphore to lock.

flags

Semaphore flags.

Only two flags are currently defined:

SEM_WAIT

If the semaphore can't be locked because another task or thread currently has it locked, setting this bit will cause the current task to go to sleep until the semaphore becomes available.

SEM_SHAREDREAD

This lets you lock the semaphore in shared mode. Multiple clients can lock a semaphore in shared mode, but only one client can lock it in exclusive mode. If you're locking the semaphore to simply read data, you should use this flag. If you are locking the semaphore to modify data, you should NOT use this flag.

Return Value

Returns 1 if the semaphore was successfully locked, or 0 if the semaphore is already locked and the SEM_WAIT flag was not set. Returns a negative error code for failure.

Implementation

Folio call implemented in kernel folio V20.

Associated Files

<:kernel:semaphore.h>, libc.a

See Also

FindSemaphore(), UnlockSemaphore()

UnlockSemaphore

Unlocks a semaphore.

Synopsis

```
Err UnlockSemaphore( Item s )
```

Description

This function unlocks the specified semaphore, allowing other task to gain access to the semaphore.

Arguments

s

The item number of the semaphore to unlock.

Return Value

Returns 0 if successful or a negative error code for failure. Possible error codes currently include:

BADITEM

The item number supplied does not refer to a valid semaphore item.

NOTOWNER

The semaphore specified by the s argument is not owned by the task.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:semaphore.h>, libc.a

See Also

LockSemaphore()

AllocSignal

Allocates signals.

Synopsis

```
int32 AllocSignal(int32 sigMask);
```

Description

One of the ways tasks communicate is by sending signals to each other. Signals are 1-bit flags that indicate that a particular event has occurred.

Tasks that send and receive signals must agree on which signal bits to use and the meanings of the signals. Except for system signals, there are no conventions for the meanings of individual signal bits; it is up to software developers to define their meanings.

You allocate bits for new signals by calling `AllocSignal()`. To allocate a single signal bit, you can do:

```
theSignal = AllocSignal(0);
```

This allocates the next unused bit in the signal word. In the return value, the bit that was allocated is set. If the allocation fails (which happens if all the non-reserved bits in the signal word are already allocated), the function returns 0.

In rare cases, you may need to define more than one signal with a single call. You do this by creating a 32-bit mask and setting any bits you want to allocate for new signals, then calling `AllocSignal()` with this bitmask as argument. If all the signals are successfully allocated, the bits set in the return value are the same as the bits that were set in the argument.

Signals are implemented as follows:

- * Each task has a 32-bit signal mask that specifies the signals it understands. Tasks allocate bits for new signals by calling `AllocSignal()`. The bits are numbered from 0 (the least-significant bit) to 31 (the most-significant bit). Bits 0 through 7 are reserved for system signals (signals sent by the kernel to all tasks); remaining bits can be allocated for other signals. Bit 31 is also reserved for the system. It is set when the kernel returns an error code to a task instead of signals. For example, trying to allocate a system signal or signal number 31.
- * A task calls `SendSignal()` to send one or more signals to another task. Each bit set in the `signalWord` argument specifies a signal to send. Normally, only one signal is sent at a time.
- * When `SendSignal()` is called, the kernel gets the incoming signal word and ORs it into the received signal mask of the target task. If the task was in wait state, it compares the received signals with the mask of waited signals. If there are any bits set in the target, the task is moved to the ready queue where it awaits attention from the CPU.
- * A task gets incoming signals by calling `WaitSignal()`. If any bits matching the supplied mask are set in the task's signal word, `WaitSignal()` returns immediately. If no matching bits are set in

the task's received signal word, the task enters wait state until a signal arrives that matches one of the signals the task is waiting for.

The system signals currently include:

SIGF_IODONE

 Informs the task that an asynchronous I/O request is complete.

SIGF_DEADTASK

 Informs the task that one of its child tasks or threads has been deleted.

Arguments

 signalMask

 An bitmask in which the bits to be allocated are set, or 0 to allocate the next available bit.
 You should use 0 whenever possible.

Return Value

 The function returns bitmask value with bits set for any bits that were allocated or 0 if not all of the requested bits could be allocated. It returns ILLEGALSIGNAL if the signalMask specified a reserved signal.

Implementation

 Folio call implemented in kernel folio V20.

Associated Files

 <:kernel:task.h>, libc.a

See Also

 FreeSignal(), GetCurrentSignals(), SendSignal(), WaitSignal(),
 ClearCurrentSignals()

ClearCurrentSignals

Clears some received signal bits.

Synopsis

```
Err ClearCurrentSignals(int32 sigMask);
```

Description

This macro resets the requested signal bits of the current task to 0.

Arguments

sigMask
A 32-bit word indicating which signal bits should be cleared.

Return Value

Returns ≥ 0 for success or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:task.h>` V24.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`AllocSignal()`, `FreeSignal()`, `SendSignal()`, `WaitSignal()`,
`GetCurrentSignals()`

FreeSignal

Frees signals.

Synopsis

```
Err FreeSignal(int32 sigMask);
```

Description

This function frees one or more signal bits allocated by `AllocSignal()`. The freed bits can then be reallocated.

Arguments

`sigMask`

A bitmask in which any signal bits to deallocate are set.

Return Value

Returns ≥ 0 if the signal(s) were freed, or a negative error code for failure.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`AllocSignal()`, `WaitSignal()`, `SendSignal()`, `GetCurrentSignals()`
`ClearCurrentSignals()`

GetCurrentSignals

Gets the currently received signal bits.

Synopsis

```
int32 GetCurrentSignals(void);
```

Description

This macro returns the signal bits that have been received by the current task.

Return Value

A 32-bit word in which all currently received signal bits are set.

Implementation

Macro implemented in `<:kernel:task.h>` V20.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`AllocSignal()`, `FreeSignal()`, `SendSignal()`, `WaitSignal()`,
`ClearCurrentSignals()`

GetTaskSignals

Gets the currently received signal bits for a task.

Synopsis

```
int32 GetTaskSignals(Task *t);
```

Description

This macro returns the signal bits that have been received by the specified task.

Return Value

A 32-bit word in which all currently received signal bits for the task are set.

Implementation

Macro implemented in `<:kernel:task.h>` V21.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`AllocSignal()`, `FreeSignal()`, `SendSignal()`, `WaitSignal()`

SendSignal

Sends signals to another task.

Synopsis

```
Err SendSignal(Item task, int32 sigMask);
```

Description

This function sends one or more signals to the specified task.

Arguments

task

The item number of the task to send signals to. If this parameter is 0, then the signals are sent to the calling task. This is sometimes useful to set initial conditions.

sigMask

The signals to send.

Return Value

The function returns 0 if successful or a negative error code for failure. Possible error codes currently include:

BADPRIV

The task attempted to send a system signal.

ILLEGALSIGNAL

The task attempted to send a signal to a task that was not allocated by that task, or bit 31 in the sigMask argument (which is reserved) was set.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:task.h>, libc.a

See Also

AllocSignal(), FreeSignal(), GetCurrentSignals(), WaitSignal()

WaitSignal

Waits until any of a set of signals are received.

Synopsis

```
int32 WaitSignal(int32 sigMask);
```

Description

This function puts the calling task into wait state until any of the signals specified in sigMask have been received. When a task is in wait state, it uses no CPU time.

When WaitSignal() returns, bits set in the result indicate which of the signals the task was waiting for were received since the last call to WaitSignal(). If the task was not waiting for certain signals, the bits for those signals remain set in the task's signal word, and all other bits in the signal word are cleared.

See AllocSignal() for a description of the implementation of signals.

Arguments

sigMask

A mask in which bits are set to specify the signals the task wants to wait for.

Return Value

Returns a mask that specifies which of the signals a task was waiting for have been received, or a negative error code for failure. Possible error codes currently include:

ILLEGALSIGNAL

One or more of the signal bits in the sigMask argument were not allocated by the task.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:task.h>, libc.a

Notes

Because it is possible for tasks to send signals in error, it is up to tasks to confirm that the actual event occurred when they receive a signal.

For example, if you were waiting for SIGF_IODONE and the return value from WaitSignal() indicated that the signal was sent, you should still call CheckIO() using the IOReq to make sure it is actually done. If it was not done you should go back to WaitSignal().

See Also

AllocSignal(), SendSignal(), GetCurrentSignals(), ClearCurrentSignals()

ConvertFP_TagData

Store a floating point value in a TagArg.

Synopsis

```
TagData ConvertFP_TagData (float32 a);
```

Description

Prepares a floating point value for storing in a TagArg.

Because TagData is defined as a pointer, casting a floating point to TagData would cause a compiler error or truncation, neither of which is the desired result. This function simply changes the type of the floating point value without changing its contents. The function ConvertTagData_FP() reverses the process.

Arguments

a
floating point value to store in a TagArg

Return Value

Floating point value converted to a TagData.

Implementation

Link library call implemented in libc.a V27.

Examples

```
void dosomething (Item instrument, float32 amplitude)
{
    StartInstrumentVA (instrument,
        AF_TAG_AMPLITUDE_FP, ConvertFP_TagData(amplitude),
        TAG_END);
    .
    .
    .
}
```

Caveats

If you don't use this function when filling out a TagArg with a floating point value, you will get a compiler error. If you forget to use it when passing a floating point value to a variable argument tag function (e.g. StartInstrumentVA()), you will get a corrupted tag list without any compile time warning.

Associated Files

<:kernel:tags.h>, libc.a

See Also

ConvertTagData_FP()

ConvertTagData_FP

Extract a floating point value stored in a TagArg.

Synopsis

```
float32 ConvertTagData_FP (TagData a);
```

Description

Extracts a floating point value stored in a TagArg.

Arguments

a
TagData value from ta_Arg of a TagArg.

Return Value

Floating point value stored in a.

Implementation

Link library call implemented in libc.a V27.

Examples

```
void dosomething (const TagArg *tags)
{
    float32 amplitude = ConvertTagData_FP (GetTagData (tags,
        AF_TAG_AMPLITUDE_FP,
        ConvertFP_TagData(0.0)));
    .
    .
    .
}
```

Associated Files

<:kernel:tags.h>, libc.a

See Also

ConvertFP_TagData()

DumpTagList

Prints the contents of a tag list.

Synopsis

```
void DumpTagList(const TagArg *tagList, const char *desc);
```

Description

This function prints out the contents of a TagArg list to the debugging terminal.

Arguments

tagList
The list of tags to print.

desc
Description of tag list to print. Can be NULL.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:tags.h>, libc.a

See Also

NextTagArg(), GetTagArg()

FindTagArg

Looks through a tag list for a specific tag.

Synopsis

```
TagArg *FindTagArg(const TagArg *tagList, uint32 tag);
```

Description

This function scans a tag list looking for a TagArg structure with a ta_Tag field equal to the tag parameter. The function always returns the last TagArg structure in the list which matches the tag. Finally, this function deals with the various control tags such as TAG_JUMP and TAG_NOP.

Arguments

tagList The list of tags to scan.

tag The value to look for.

Return Value

Returns a pointer to a TagArg structure with a value of ta_Tag that matches the tag parameter, or NULL if no match can be found. The function always returns the last tag in the list which matches.

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:tags.h>, libc.a

See Also

NextTagArg()

GetTagArg

Finds a TagArg in list and returns its ta_Arg field.

Synopsis

```
TagData GetTagArg(const TagArg *tagList, uint32 tag,
                  TagData defaultValue);
```

Description

This function calls FindTagArg() to locate the specified tag. If it is found, it returns the ta_Arg value from the found TagArg. Otherwise, it returns the default value supplied. This is handy when resolving a tag list that has optional tags that have suitable default values.

Arguments

tagList
The list of tags to scan. Can be NULL.

tag
The tag ID to look for.

defaultValue
Default value to use when specified tag isn't found in tagList.

Return Value

ta_Arg value from found TagArg or defaultValue.

Implementation

Link library call implemented in libc.a V24.

Examples

```
void dosomething (const TagArg *tags)
{
    uint32 amplitude = (uint32)GetTagData (tags, MY_TAG_AMPLITUDE,
    (TagData)0x7fff);
    .
    .
    .
}
```

Caveats

It's a good idea to always use casts for the default value and result. Don't assume anything about the type definition of TagData other than that it is a 32-bit value.

Associated Files

<:kernel:tags.h>, libc.a

See Also

FindTagArg(), NextTagArg()

NextTagArg

Finds the next TagArg in a tag list.

Synopsis

```
TagArg *NextTagArg(const TagArg **tagList);
```

Description

This function iterates through a tag list, skipping and chaining as dictated by control tags. There are three control tags:

TAG_NOP

Ignores that single entry and moves to the next one.

TAG_JUMP

Has a pointer to another array of tags.

TAG_END

Marks the end of the tag list.

This function only returns TagArgs which are not system tags. Each call returns either the next TagArg you should examine, or NULL when the end of the list has been reached.

Arguments

tagList

This is a pointer to a storage location used by the iterator to keep track of its current location in the tag list. The variable that this parameter points to should be initialized to point to the first TagArg in the tag list, and should not be changed thereafter.

Return Value

Returns a pointer to a TagArg structure, or NULL if all the tags have been visited. None of the control tags are ever returned to you, they are handled transparently by this function.

Example

```
void WalkTagList(const TagArg *tags)
{
    TagArg *state;
    TagArg *tag;

    state = tags;
    while ((tag = NextTagArg(&state)) != NULL)
    {
        switch (tag->ta_Tag)
        {
            case TAG1: // process this tag
                break;

            case TAG2: // process this tag
                break;

            default : // unknown tag, return an error
                break;
        }
    }
}
```

```
}
```

Implementation

Folio call implemented in Kernel folio V24.

Associated Files

<:kernel:tags.h>, libc.a

See Also

FindTagArg()

CreateModuleThread

Creates a thread from a loaded code module.

Synopsis

```
Item CreateModuleThread(Item module, const char *name,  
                        const TagArg *tags);  
  
Item CreateModuleThreadVA(Item module, const char *name,  
                        uint32 tags, ...);
```

Description

This function creates a thread. Threads provide a preemptively scheduled execution context which has all of the characteristics of a full-fledged task, except that they don't have their own address space, and therefore share the one of their parent

This function launches a loaded code module as a thread. You can also launch a function within the current code module by using `CreateThread()`.

The stack size and priority for the new thread are determined by the values specified to the linker when the module was linked. If no priority was given at link time, the thread will be launched with the priority of the current task.

When you no longer need a thread, use `DeleteModuleThread()` to delete it. Alternatively, the thread can return or call `exit()`.

Arguments

- module**
The loaded code module to execute as a thread, as obtained from `OpenModule()`
- name**
The name of the thread to be created. You can later use `FindTask()` to find it by name.
- tags**
A pointer to an array of optional tag arguments containing extra data for this function, or NULL. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with `TAG_END`.

CREATETASK_TAG_ARGC (uint32)
A 32-bit value that will be passed to the thread being launched as `argc` for its `main()` function. If this tag is omitted, `argc` will be 0.

CREATETASK_TAG_ARGP (uint32)
A 32-bit value that will be passed to the thread being launched as `argv` for its `main()` function. If this tag is omitted, `argv` will be NULL.

CREATETASK_TAG_MSGFROMCHILD (Item)
Provides the item number of a message port. The kernel will send a status message to this port whenever the thread being created exits. The message is sent by the kernel after the thread has been deleted. The `msg_Result` field of the message contains the exit status of the thread. This is the value the thread provided to `exit()`, or the value returned by the

thread's `main()` function. The `msg_Val1` field of the message contains the item number of the thread that just terminated. Finally, the `msg_Val2` field contains the item number of the task or thread that terminated the thread. If the thread exited on its own, this will be the item number of the thread itself. It is the responsibility of the task that receives the status message to delete it when it is no longer needed by using `DeleteMsg()`.

CREATETASK_TAG_MAXQ (uint32)

A value indicating the maximum quanta for the thread in microseconds.

CREATETASK_TAG_USERDATA (void *)

This specifies an arbitrary 32-bit value that is put in the new thread's `t_UserData` field. This is a convenient way to pass a pointer to a shared data structure when starting a thread.

CREATETASK_TAG_USEREXCHANDLER (UserHandler)

Lets you specify the exception handler for the thread.

CREATETASK_TAG_DEFAULTMSGPORT (void)

When this tag is present, the kernel automatically creates a message port for the new thread being started. The item number of this port is stored in the Task structure's `t_DefaultMsgPort` field. This is a convenient way to quickly establish a communication channel between a parent and a child.

Return Value

Returns the item number of the thread or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V27`.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`DeleteModuleThread()`, `exit()`, `OpenModule()`, `system()`

CreateTask

Creates a task from a loaded code module.

Synopsis

```
Item CreateTask(Item module, const char *name, const TagArg *tags);
Item CreateTaskVA(Item module, const char *name, uint32 tags, ...);
```

Description

This function launches a loaded code module as a task.

The stack size and priority for the new task are determined by the values specified to the linker when the module was linked. If no priority was given at link time, the task will be launched with the priority of the current task.

When you no longer need a task, use `DeleteTask()` to delete it. Alternatively, the task can return or call `exit()`.

Arguments

module

The loaded code module to launch as a task, as obtained from `OpenModule()`

name

The name of the task to be created. You can later use `FindTask()` to find it by name.

tags

A pointer to an array of optional tag arguments containing extra data for this function, or `NULL`. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with `TAG_END`.

`CREATETASK_TAG_ARGC` (uint32)

A 32-bit value that will be passed to the task being launched as `argc` for its `main()` function. If this tag is omitted, `argc` will be 0.

`CREATETASK_TAG_ARGP` (uint32)

A 32-bit value that will be passed to the task being launched as `argv` for its `main()` function. If this tag is omitted, `argv` will be `NULL`.

`CREATETASK_TAG_CMDSTR` (char *)

A pointer to a string to use to build an `argv[]`-style array to pass in to the task being launched for its `main()` function. Using this tag overrides values you might have supplied with `CREATETASK_TAG_ARGC` or `CREATETASK_TAG_ARGP`.

`CREATETASK_TAG_MSGFROMCHILD` (Item)

Provides the item number of a message port. The kernel will send a status message to this port whenever the task being created exits. The message is sent by the kernel after the task has been deleted. The `msg_Result` field of the message contains the exit status of the task. This is the value the task provided to `exit()`, or the value returned by the task's `main()` function. The `msg_Val1` field of the message contains the item number of the task that just terminated. Finally, the `msg_Val2` field contains the item number of the thread or task that terminated the task. If the task exited on its own, this will be the item number

of the task itself. It is the responsibility of the task that receives the status message to delete it when it is no longer needed by using `DeleteMsg()`.

`CREATETASK_TAG_MAXQ` (uint32)

A value indicating the maximum quanta for the task in microseconds.

`CREATETASK_TAG_USERDATA` (void *)

This specifies an arbitrary 32-bit value that is put in the new task's `t_UserData` field. This is a convenient way to pass a pointer to a shared data structure.

`CREATETASK_TAG_USEREXCHANDLER` (UserHandler)

Lets you specify the exception handler for the task.

`CREATETASK_TAG_DEFAULTMSGPORT` (void)

When this tag is present, the kernel automatically creates a message port for the new task being started. The item number of this port is stored in the Task structure's `t_DefaultMsgPort` field. This is a convenient way to quickly establish a communication channel between a parent and a child.

Return Value

Returns the item number of the task or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V27`.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`DeleteTask()`, `exit()`, `OpenModule()`, `system()`

CreateThread

Creates a thread.

Synopsis

```
Item CreateThread(void (*code)(), const char *name, uint8 pri,  
                  int32 stackSize, const TagArg *tags);
```

```
Item CreateThreadVA(void (*code)(), const char *name, uint8 pri,  
                   int32 stackSize, uint32 tags, ...);
```

Description

This function creates a thread. Threads provide a preemptively scheduled execution context which has all of the characteristics of a full-fledged task, except that they don't have their own address space, and therefore share the one of their parent task.

There is no default size for a thread's stack. To avoid stack overflow errors, the stack must be large enough to handle any possible uses. One way to find the proper stack size for a thread is to start with a very large stack, reduce its size until a stack overflow occurs, and then double its size. The memory for the stack is allocated by this function and gets freed automatically when the thread exits.

This function takes a function pointer and the thread will begin execution at that point in memory. You can also load in some external code and run it as a thread using `CreateModuleThread()`.

When you no longer need a thread, use `DeleteThread()` to delete it. Alternatively, the thread can return or call `exit()`.

Arguments

code

A pointer to the code that the thread executes.

name

The name of the thread to be created. You can later use `FindTask()` to find it by name.

pri

The priority of the thread, in the range 11 to 199. A larger number specifies a higher priority. A value of 0 for this argument specifies that the thread should be launched at the same priority as the current task.

stackSize

The size in bytes of the thread's stack. A good default value for this is 4096.

tags

A pointer to an array of optional tag arguments containing extra data for this function, or NULL. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

CREATETASK_TAG_ARGC (uint32)

A 32-bit value that will be passed to the thread being launched as its first argument. If this is omitted, the first argument will be 0.

CREATETASK_TAG_ARGP (uint32)

A 32-bit value that will be passed to the thread being launched as a second argument. If this is omitted, the second argument will be 0.

CREATETASK_TAG_MSGFROMCHILD (Item)

Provides the item number of a message port. The kernel will send a status message to this port whenever the thread being created exits. The message is sent by the kernel after the thread has been deleted. The msg_Result field of the message contains the exit status of the thread. This is the value the task provided to exit(), or the value returned by the thread initial function. The msg_Val1 field of the message contains the item number of the thread that just terminated. Finally, the msg_Val2 field contains the item number of the thread or task that terminated the thread. If the thread exited on its own, this will be the item number of the thread itself. It is the responsibility of the task that receives the status message to delete it when it is no longer needed by using DeleteMsg().

CREATETASK_TAG_MAXQ (uint32)

A value indicating the maximum quanta for the thread in microseconds.

CREATETASK_TAG_USERDATA (void *)

This specifies an arbitrary 32-bit value that is put in the new thread's t_UserData field. This is a convenient way to pass a pointer to a shared data structure when starting a thread.

CREATETASK_TAG_USEREXCHANDLER (UserHandler)

Lets you specify the exception handler for the thread.

CREATETASK_TAG_DEFAULTMSGPORT (void)

When this tag is present, the kernel automatically creates a message port for the new thread being started. The item number of this port is stored in the Task structure's t_DefaultMsgPort field. This is a convenient way to quickly establish a communication channel between a parent and a child.

Return Value

Returns the item number of the thread or a negative error code for failure.

Implementation

Link library call implemented in libc.a V27.

Associated Files

<:kernel:task.h>, libc.a

See Also

DeleteThread(), exit()

DeleteModuleThread Deletes a thread.

Synopsis

```
Err DeleteModuleThread(Item thread);
```

Description

This function deletes a thread. Any items owned by the thread are automatically freed. Memory allocated by the thread is NOT freed however and remains the property of the parent task.

Arguments

thread
The item number of the thread to be deleted.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Macro implemented implemented in `<:kernel:task.h>` V27.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`CreateModuleThread()`, `exit()`

DeleteTask

Deletes a task.

Synopsis

```
Err DeleteTask(Item task);
```

Description

This function deletes a task. Any items owned by the task, which includes any threads of that task, are automatically freed. Any memory allocated by the task is also freed.

Arguments

`task`

The item number of the task to be deleted.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Macro implemented implemented in `<:kernel:task.h>` V27.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`CreateTask()`, `exit()`

DeleteThread

Deletes a thread.

Synopsis

```
Err DeleteThread(Item thread);
```

Description

This function deletes a thread. Any items owned by the thread are automatically freed. Memory allocated by the thread is NOT freed however and remains the property of the parent task.

Arguments

thread

The item number of the thread to be deleted.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Macro implemented implemented in `<:kernel:task.h>` V27.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`CreateThread()`, `exit()`

exit

Exits from a task or thread.

Synopsis

```
void exit( int status )
```

Description

This procedure deletes the calling task or thread. If the `CREATETASK_TAG_MSGFROMCHILD` tag was set when creating the calling task, then the status is sent to the parent process through a message.

Arguments

status

The status to be returned to the parent of the calling task or thread. Negative status is reserved for system use.

Return Value

This procedure never returns.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

`<stdlib.h>`, `libc.a`

Notes

When tasks (including threads) have finished their work, they should call `exit()` to die gracefully. The system will call `exit()` on behalf of the task that returns from the top-level routine. `exit()` does necessary clean-up work when a task is finished, including the cleanup required for a thread created by the `CreateThread()` library routine.

See Also

`CreateThread()`, `DeleteItem()`, `DeleteThread()`

FindTask

Finds a task by name.

Synopsis

```
Item FindTask(const char *name);
```

Description

This macro finds a task with the specified name. The search is not case-sensitive.

To get a pointer to the current task, use the CURRENTTASK macro defined in `<:kernel:task.h>`

Arguments

name
The name of the task to find.

Return Value

Returns the item number of the task that was found, or a negative error code for failure.

Implementation

Macro implemented in `<:kernel:task.h>` V20.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`CreateThread()`

InvalidateFPState

Invalidates the current contents of the floating-point registers to prevent them from being saved during a context switch.

Synopsis

```
void InvalidateFPState(void);
```

Description

Portfolio tries to handle floating-point registers in a smart way to improve overall system performance. This function can be used by savvy clients to improve multitasking speed.

When performing a task switch operation, the state of the FP registers is not saved along with the regular registers. Instead, Portfolio has the concept of a floating-point owner task. If a task attempts to do a floating-point operation without being the current owner of the FPU, the kernel automatically saves the FP state of the current owner, loads the FP state of the task attempting to do an FP operation, and then marks this task as the FPU owner.

The reason behind this scheme is to keep to a minimum the number of times FP state needs to be saved and restored. For example, if thread A does floating-point calculations, and thread B starts running and doesn't do any floating-point, and thread A starts running again, there will have been no FP save/restore operation performed. This saves a considerable amount of time which improves effective task switch performance.

As a result of this architecture, you will get optimal performance if only one of your threads does FP operations, and all the others are purely integer based. With such a setup, no FP state will ever need to be saved or restored.

This function is intended for savvy clients that realize at some point that the current contents of the FP registers is no longer of any value, and so does not need to be preserved. Making this call effectively invalidates the contents of the FP registers.

Implementation

Folio call implemented in Kernel folio V27.

RegisterUserException

Registers the exceptions to be handled by the current task.

Synopsis

```
Err RegisterUserException(uint32 excmask, uint32 flag);
```

Description

Determines which exceptions are handled by the task's exception handler, and which should be handled by the kernel.

Arguments

excmask

A bitmask of the exceptions to affect in this call.

flag

A flag value of 1 specifies the task wants to handle the exceptions set in excmask. A flag value of 0 specifies the task does not want to deal with the exceptions set in excmask.

Additionally, for floating point exceptions, the flag must be or-ed with the types of floating point exceptions to affect in this call.

Return Value

Returns ≥ 0 for success or a negative error code for failure. Possible error codes currently include:

ER_ParamError

The flag argument is invalid.

Implementation

Folio call implemented in Kernel folio V27.

Associated Files

<:kernel:task.h>, libc.a

See Also

RegisterUserExcHandler()

RegisterUserExcHandler

Registers an exception handler for the current task.

Synopsis

```
Err RegisterUserExcHandler(UserHandler exchandler);
```

Description

As a task executes, it may cause exceptions to occur. When this happens, the kernel's response is normally to remove the offending task from the system and continues on with regular business.

Using this function, a task can register a handler that the kernel will call when specific kinds of exceptions occur. This handler can then correct what caused the exception and resume normal task execution, or it can perform last minute cleanup before the task is deleted by the kernel.

You can also install an exception handler for a task when the task is first created by using the `CREATETASK_TAG_USEREXCHANDLER` tag.

The exception handler is invoked when any of the exceptions previously registered using `RegisterUserException()` occur. The handler is called with an argument specifying the type of exception that occurred, in addition to additional exception-specific arguments.

The `EXC_FP` exception occurs when floating-point operations cause the CPU to raise an exception. The handler is called with the first parameter being `EXC_FP`, the second parameter being a pointer to a `RegBlock` structure, and the third parameter being a pointer to an `FPRegBlock` structure. The `RegBlock` and `FPRegBlock` structures reflect the exact state of the processor when the exception occurred. You can modify these structures at will. When your handler returns 0, the kernel will use the contents of the `RegBlock` and `FPRegBlock` as register value for the task. You can therefore alter register values of your mainline code. You need to, in the exception handler, reset the status bits in `FPSCR` register in `FPRegBlock` structure after handling the corresponding exceptions.

The `EXC_AT_EXIT` exception is raised when the current task is being deleted. The first parameter supplied to the handler is `EXC_AT_EXIT`, the second parameter is the exit status of the task, and the third parameter is `NULL`. The return value of the exception handler is ignored for this handler, the task will always be deleted.

The exception handler must return 0 after successfully handling the exception. If it returns a negative error code, the kernel's default exception handler is invoked, which will usually lead to the current task being deleted.

Arguments

exchandler

This is the task-specific user exception handler.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

`<:kernel:task.h>`, `libc.a`

See Also

`RegisterUserException()`

Yield

Give up the CPU to a task of equal priority.

Synopsis

```
void Yield( void )
```

Description

In Portfolio, high-priority tasks always have precedence over lower priority tasks. Whenever a high priority task becomes ready to execute, it will instantly interrupt lower priority tasks and start running. The lower priority tasks do not get to finish their time quantum, and just get put into the system's ready queue for future scheduling.

If there are a number of tasks of equal priority which are all ready to run, the kernel does round-robin scheduling of these tasks. This is a process by which each task is allowed to run for a fixed amount of time before the CPU is taken away from it, and given to another task of the same priority. The amount of time a task is allowed to run before being preempted is called the task's "quantum".

The purpose of the Yield() function is to let a task voluntarily give up the remaining time of its quantum. Since the time quantum is only an issue when the kernel does round-robin scheduling, it means that Yield() actually only does something when there are multiple ready tasks at the same priority. However, since the yielding task does not know exactly which task, if any, is going to run next, Yield() should not be used for implicit communication amongst tasks. The way to cooperate amongst tasks is using signals, messages, and semaphores.

In short, if there are higher-priority tasks in the system, the current task will only run if the higher-priority tasks are all in the wait queue. If there are lower-priority tasks, these will only run if the current task is in the wait queue. And if there are other tasks of the same priority, the kernel automatically cycles through all the tasks, spending a quantum of time on each task, unless a task calls Yield(), which will cut short its quantum.

If there are no other ready tasks of the same priority as the task that calls Yield(), then that task will keep running as if nothing happened.

Yield() is only useful in very specific circumstances. If you plan on using it, think twice. In most cases it is not needed, and using it will result in a general degradation of system performance.

Implementation

Folio call implemented in Kernel folio V20.

Associated Files

<:kernel:task.h>, libc.a

See Also

`WaitSignal()`

AddTimerTicks

Adds two TimerTicks values together.

Synopsis

```
void AddTimerTicks(const TimerTicks *tt1, const TimerTicks *tt2,  
                  TimerTicks *result);
```

Description

Adds two hardware-dependant timer ticks values together.

Arguments

tt1

The first time value to add.

tt2

The second time value to add.

result

A pointer to the location where the resulting time value will be stored. This pointer can match either tt1 or tt2.

Implementation

Folio call implemented in kernel folio V27.

Warning

Do not make any assumptions about the internal representation of the TimerTicks structure. Only use the supplied functions to manipulate the structure.

Associated Files

<:kernel:time.h>, libc.a

See Also

SubTimerTicks(), CompareTimerTicks()

AddTimes

Adds two time values together.

Synopsis

```
void AddTimes(const TimeVal *tv1, const TimeVal *tv2,  
              TimeVal *result);
```

Description

Adds two time values together, yielding the total time for both.

Arguments

tv1

The first time value to add.

tv2

The second time value to add.

result

A pointer to the location where the resulting time value will be stored. This pointer can match either tv1 or tv2.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:time.h>, libc.a

See Also

SubTimes(), CompareTimes()

CompareTimerTicks

Compares two timer ticks values.

Synopsis

```
int32 CompareTimerTicks(const TimerTicks *tt1, const TimerTicks
*tt2);
```

Description

Compares two hardware-dependant timer ticks values to determine which came first.

Arguments

`tt1`
The first time value.

`tt2`
The second time value.

Return Value

< 0
if (tt1 < tt2)

0
if (tt1 == tt2)

> 0
if (tt1 > tt2)

Implementation

Folio call implemented in kernel folio V27.

Warning

Do not make any assumptions about the internal representation of the `TimerTicks` structure. Only use the supplied functions to manipulate the structure.

Associated Files

<:kernel:time.h>, libc.a

See Also

`AddTimerTicks()`, `SubTimerTicks()`, `TimerTicksLaterThan()`,
`TimerTicksLaterThanOrEqual()`

CompareTimes

Compares two time values.

Synopsis

```
int32 CompareTimes(const TimeVal *tv1, const TimeVal *tv2);
```

Description

Compares two time values to determine which came first.

Arguments

tv1
The first time value.

tv2
The second time value.

Return Value

< 0
if (tv1 < tv2)

0
if (tv1 == tv2)

> 0
if (tv1 > tv2)

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:time.h>, libc.a

See Also

AddTimes(), SubTimes(), TimeLaterThan(), TimeLaterThanOrEqual()

ConvertTimerTicksToTimeVal

Convert a hardware dependant timer tick value to a TimeVal.

Synopsis

```
void ConvertTimerTicksToTimeVal(const TimerTicks *tt, TimeVal *tv);
```

Description

This function converts a hardware-dependant representation of system time to a TimeVal structure.

Do not assume **anything** about the value stored in the TimerTicks structure. The meaning and interpretation will change based on the CPU performance, and possibly other issues. Only use the supplied functions to operate on this structure.

Arguments

tt

The timer tick value to convert.

tv

A pointer to a TimeVal structure which will receive the converted value.

Implementation

Folio call implemented in kernel folio V27.

Warning

Do not make any assumptions about the internal representation of the TimerTicks structure. Only use the supplied functions to manipulate the structure.

Associated Files

<:kernel:time.h>, libc.a

See Also

ConvertTimeValToTimerTicks()

ConvertTimeValToTimerTicks

Convert a time value to a hardware dependant form.

Synopsis

```
void ConvertTimeValToTimerTicks(const TimeVal *tv, TimerTicks *tt);
```

Description

This function converts a TimeVal structure to a hardware-dependant representation.

Do not assume **anything** about the value stored in the TimerTicks structure. The meaning and interpretation will change based on the CPU performance, and possibly other issues. Only use the supplied functions to operate on this structure.

Arguments

tv

The time value to convert.

tt

A pointer to a TimerTicks structure which will receive the converted value.

Implementation

Folio call implemented in kernel folio V27.

Warning

Do not make any assumptions about the internal representation of the TimerTicks structure. Only use the supplied functions to manipulate the structure.

Associated Files

<:kernel:time.h>, libc.a

See Also

ConvertTimerTicksToTimeVal()

CreateTimerIOReq

Creates a timer device I/O request.

Synopsis

```
Item CreateTimerIOReq(void);
```

Description

Creates an I/O request for communication with the timer device.

Return Value

Returns a timer I/O request item, or a negative error code for failure.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:time.h>, libc.a

See Also

DeleteTimerIOReq(), WaitTime(), WaitUntil()

DeleteTimerIOReq

Delete a timer device I/O request.

Synopsis

```
Err DeleteTimerIOReq(Item ioreq);
```

Description

Frees any resources used by a previous call to CreateTimerIOReq().

Arguments

`ioreq`
The I/O request item, as returned by a previous call to CreateTimerIOReq().

Return Value

Return ≥ 0 if successful, or a negative error code for failure.

Implementation

Link library call implemented in libc.a V24.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

CreateTimerIOReq(), WaitTime(), WaitUntil()

SampleSystemTimeTT

Samples the system time with very low overhead.

Synopsis

```
void SampleSystemTimeTT( TimerTicks *time )
```

Description

This function records the current system time in a hardware dependant format. This is an extremely low overhead call giving a very high-accuracy timing.

You can convert the hardware dependant TimerTicks structure to a more abstract form by using the ConvertTimerTicksToTimeVal() function.

Do not assume *anything* about the value stored in the TimerTicks structure. The meaning and interpretation will change based on the CPU performance, and possibly other issues. Only use the supplied functions to operate on this structure.

Arguments

time

A pointer to a TimerTicks structure which will receive the current system time.

Implementation

Folio call implemented in kernel folio V27.

Warning

Do not make any assumptions about the internal representation of the TimerTicks structure. Only use the supplied functions to manipulate the structure.

Associated Files

<:kernel:time.h>, libc.a

See Also

ConvertTimerTicksToTimeVal()

SampleSystemTimeTV

Samples the system time with very low overhead.

Synopsis

```
void SampleSystemTimeTV( TimeVal *time )
```

Description

This function records the current system time in the supplied TimeVal structure. This is a very low overhead call giving a very high-accuracy timing.

The time value returned by this function corresponds to the time maintained by the TIMER_UNIT_USEC unit of the timer device.

For an even faster routine, see SampleSystemTimeTT(), which deals in a hardware-dependant representation of time.

Arguments

time

A pointer to a TimeVal structure which will receive the current system time.

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:time.h>, libc.a

SampleSystemTimeVBL

Samples the system VBL count with very low overhead.

Synopsis

```
void SampleSystemTimeVBL( TimeValVBL *tv )
```

Description

This function records the current system VBL count. This is an

Arguments

tv
A pointer to a TimeValVBL structure which will receive the current system VBL count.

Warning

The VBlank timer runs at either 50Hz or 60Hz depending on whether the system is displaying PAL or NTSC.

Implementation

Folio call implemented in kernel folio V27.

Associated Files

<:kernel:time.h>, libc.a

StartMetronome

Start a metronome counter.

Synopsis

```
Err StartMetronome(Item ioreq, uint32 seconds, uint32 micros,  
                    int32 signal);
```

Description

Starts a metronome timer. Once this call returns, a signal will be sent to your task on a fixed interval until you call `StopMetronome()`. The signalling interval is specified with the `seconds` and `micros` argument. The signal to send is specified with the `signal` argument.

Arguments

`ioreq`
An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

`seconds`
The number of seconds between signals of the metronome.

`micros`
The number of microseconds between signals of the metronome.

`signal`
The signal mask to send whenever the requested interval of time passes.

Return Value

`>= 0` for success, or a negative error code for failure. Once the metronome is started, you will receive signals on the requested interval. To stop the metronome, you should call the `StopMetronome()` function.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `StopMetronome()`

StartMetronomeVBL

Start a metronome counter.

Synopsis

```
Err StartMetronomeVBL(Item ioreq, uint32 fields, int32 signal);
```

Description

Starts a metronome timer. Once this call returns, a signal will be sent to your task on a fixed interval until you call `StopMetronomeVBL()`. The signalling interval is specified with the `fields` argument. The signal to send is specified with the `signal` argument.

Arguments

`ioreq`

An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

`fields`

The number of VBLs between signals of the metronome.

`signal`

The signal mask to send whenever the requested interval of time passes.

Return Value

`>= 0` for success, or a negative error code for failure. Once the metronome is started, you will receive signals on the requested interval. To stop the metronome, you should call the `StopMetronomeVBL()` function.

Implementation

Link library call implemented in `libc.a V27`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `StopMetronomeVBL()`

StopMetronome

Stop a metronome counter.

Synopsis

```
Err StopMetronome(Item ioreq);
```

Description

Stops a metronome counter that was previously started with `StartMetronome()`.

Arguments

`ioreq`
An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

Return Value

`>= 0` for success, or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `StartMetronome()`

StopMetronomeVBL

Stop a metronome counter.

Synopsis

```
Err StopMetronomeVBL(Item ioreq);
```

Description

Stops a metronome counter that was previously started with `StartMetronomeVBL()`.

Arguments

`ioreq`
An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

Return Value

`>= 0` for success, or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `StartMetronomeVBL()`

SubTimerTicks

Subtracts one TimerTicks value from another.

Synopsis

```
void SubTimerTicks(const TimerTicks *tt1, const TimerTicks *tt2,  
                  TimerTicks *result);
```

Description

Subtracts one hardware-dependant timer tick value from another. The value stored corresponds to (tt2 - tt1). If the subtraction would yield a negative time value, a zero timer tick value is returned instead.

Arguments

tt1
The first time value.

tt2
The second time value.

result
A pointer to the location where the resulting time value will be stored. This pointer can match either of tt1 or tt2. The value stored corresponds to (tt2 - tt1).

Implementation

Folio call implemented in kernel folio V27.

Warning

Do not make any assumptions about the internal representation of the TimerTicks structure. Only use the supplied functions to manipulate the structure.

Associated Files

<:kernel:time.h>, libc.a

See Also

AddTimerTicks(), CompareTimerTicks()

SubTimes

Subtracts one time value from another.

Synopsis

```
void SubTimes(const TimeVal *tv1, const TimeVal *tv2, TimeVal
*result);
```

Description

Subtracts two time values, yielding the difference in time between the two.

Arguments

tv1
The first time value.

tv2
The second time value.

result
A pointer to the location where the resulting time value will be stored. This pointer can match either of tv1 or tv2. The value stored corresponds to (tv2 - tv1).

Implementation

Link library call implemented in libc.a V24.

Associated Files

<:kernel:time.h>, libc.a

See Also

AddTimes(), CompareTimes()

TimeLaterThan

Returns whether a time value comes before another.

Synopsis

```
bool TimeLaterThan(const TimeVal *tv1, const TimeVal *tv2);
```

Description

Returns whether tv1 comes chronologically after tv2.

Arguments

tv1
The first time value.

tv2
The second time value.

Return Value

TRUE
if tv1 comes after tv2

FALSE
if tv1 comes before or is the same as tv2

Implementation

Macro implemented in `<:kernel:time.h>` V24.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CompareTimes()`, `TimeLaterThanOrEqual()`

TimeLaterThanOrEqual

Returns whether a time value comes before or at the same time as another.

Synopsis

```
bool TimeLaterThanOrEqual(const TimeVal *tv1, const TimeVal *tv2);
```

Description

Returns whether tv1 comes chronologically after tv2, or is the same as tv2.

Arguments

tv1
The first time value.

tv2
The second time value.

Return Value

TRUE
if tv1 comes after tv2 or is the same as tv2

FALSE
if tv1 comes before tv2

Implementation

Macro implemented in `<:kernel:time.h>` V24.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CompareTimes()`, `TimeLaterThan()`

TimerTicksLaterThan

Returns whether a time tick value comes before another.

Synopsis

```
bool TimerTicksLaterThan(const TimerTicks *tt1, const TimerTicks
    *tt2);
```

Description

Returns whether tt1 comes chronologically after tt2.

Arguments

tt1
The first time value.

tt2
The second time value.

Return Value

TRUE
if tt1 comes after tv2

FALSE
if tt1 comes before or is the same as tt2

Implementation

Macro implemented in `<:kernel:time.h>` V27.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CompareTimerTicks()`, `TimerTicksLaterThanOrEqual()`

TimerTicksLaterThanOrEqual

Returns whether a time tick value comes before or at the same time as another.

Synopsis

```
bool TimerTicksLaterThanOrEqual(const TimerTicks *tt1,  
                                const TimerTicks *tt2);
```

Description

Returns whether tt1 comes chronologically after tv2, or is the same as tt2.

Arguments

tt1
 The first time value.

tt2
 The second time value.

Return Value

TRUE
 if tt1 comes after tt2 or is the same as tt2

FALSE
 if tt1 comes before tt2

Implementation

Macro implemented in `<:kernel:time.h>` V27.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CompareTimerTicks()`, `TimerTicksLaterThan()`

WaitTime

Waits for a given amount of time to pass.

Synopsis

```
Err WaitTime(Item ioreq, uint32 seconds, uint32 micros);
```

Description

Puts the current context to sleep for a specific amount of time.

Arguments

ioreq
An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

seconds
The number of seconds to wait for.

micros
The number of microseconds to wait for.

Return Value

Returns ≥ 0 if successful, or a negative error code if it fails.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `WaitUntil()`

WaitTimeVBL

Waits for a given number of video fields to pass.

Synopsis

```
Err WaitTimeVBL(Item ioreq, uint32 fields);
```

Description

Puts the current context to sleep until a specified number of VBLs occur.

Arguments

ioreq
An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

fields
The number of fields to wait for.

Return Value

Returns ≥ 0 if successful, or a negative error code if it fails.

Warning

The VBlank timer runs at either 50Hz or 60Hz depending on whether the system is displaying PAL or NTSC.

Implementation

Link library call implemented in `libc.a V27`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `WaitUntilVBL()`

WaitUntil

Waits for a given amount of time to arrive.

Synopsis

```
Err WaitUntil(Item ioreq, uint32 seconds, uint32 micros);
```

Description

Puts the current context to sleep until the system clock reaches a given time.

Arguments

`ioreq`
An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

`seconds`
The seconds value that the timer must reach.

`micros`
The microseconds value that the timer must reach.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Link library call implemented in `libc.a V24`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `WaitTime()`

WaitUntilVBL

Waits for a given vblank count to be reached.

Synopsis

```
Err WaitUntilVBL(Item ioreq, uint32 fields);
```

Description

Puts the current context to sleep until the system clock reaches a given time, specified in VBLs.

Arguments

`ioreq`

An active timer device I/O request, as obtained from `CreateTimerIOReq()`.

`fields`

The vblank count value that the timer must reach.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Warning

The VBlank timer runs at either 50Hz or 60Hz depending on whether the system is displaying PAL or NTSC.

Implementation

Link library call implemented in `libc.a V27`.

Associated Files

`<:kernel:time.h>`, `libc.a`

See Also

`CreateTimerIOReq()`, `DeleteTimerIOReq()`, `WaitTimeVBL()`

Chapter 15

Portfolio Item Reference

This section presents the reference documentation for the various types of items supported within Portfolio. Each item type is listed along with the tags and functions that can be used to manipulate it.

Alias

A character string for referencing the pathname to a file or a directory of files.

Description

A file alias is a character string for referencing the pathname to a file or directory of files on disk. Whenever the File folio parses pathnames, it looks for path components starting with \$, and treats the rest of the component as the name of an alias. The file folio then finds the alias of that name, and extracts a replacement string from the alias to use as the path component.

- Folio

file

Item Type

FILEALIASNODE

Create

CreateAlias()

- Delete

DeleteItem()

Attachment

The binding of a Sample or an Envelope to an Instrument or Template.

Description

An Attachment is the item which binds a Sample or an Envelope (slave Item) to a particular Instrument or Instrument Template (master Item).

An Attachment is associated with precisely one master and one slave. An Attachment is said to be an Envelope Attachment if its slave is an Envelope, or a Sample Attachment if its slave is a Sample.

Sample attachments actually come in two flavors: one for input FIFOs and another for output FIFOs, defined by the Hook to which the Sample is attached. Both kinds are considered Sample Attachments and no distinction is made between them.

A master Item can have one Envelope Attachment per Envelope hook and one Sample Attachment per Output FIFO. A master Item can have multiple Sample Attachments per Input FIFO, but only one will be selected to be played when the instrument is started. This is useful for creating multi-sample instruments, where the sample selected to be played depends on the pitch to be played.

A slave Item can have any number of Attachments made to it.

Attachments are automatically deleted when either its master or slave Item is deleted.

By default when an Attachment is deleted, the slave Item is unaffected. An Attachment created with { AF_TAG_AUTO_DELETE_SLAVE, TRUE } causes its slave to be automatically deleted when the Attachment is deleted. Because Attachments themselves are automatically deleted when either the master or slave is deleted, the auto-deletion effect can be triggered by deleting either the master or slave of such an attachment.

Attachments made to a Template are automatically propagated to Instruments created from that Template. All properties are copied from Template Attachments to Instrument Attachments except for the AF_TAG_AUTO_DELETE_SLAVE, which is set to FALSE. AF_TAG_AUTO_DELETE_SLAVE isn't propagated because the propagation duplicates just the Template's Attachment items, not the slave items. This permits deleting instruments created from such a template without deleting the slave items attached to the template.

Folio

audio

Item Type

AUDIO_ATTACHMENT_NODE

Create

CreateAttachment()

Delete

DeleteAttachment()

Query

GetAudioItemInfo()

Modify

SetAudioItemInfo()

Use

LinkAttachments(), MonitorAttachment(), ReleaseAttachment(),
StartAttachment(), StopAttachment(), WhereAttachment()

Tags

AF_TAG_AUTO_DELETE_SLAVE (bool) - Create,
Set to TRUE to cause the slave Item to be automatically
deleted when the Attachment Item is deleted. Otherwise the
slave Item is left intact when the Attachment Item is
deleted. Defaults to FALSE.

There is no harm in setting this flag for multiple
Attachments to the same slave, nor is there any harm for
manually deleting the slave of such an Attachment.

AF_TAG_CLEAR_FLAGS (uint32) - Modify
Set of AF_ATT_ flags to clear. Clears every flag for which
a 1 is set in ta_Arg.

AF_TAG_MASTER (Item) - Create, Query
Instrument or Template item to attach envelope or sample to.
Must be specified when creating an Attachment.

AF_TAG_NAME (const char *) - Create, Query
The name of the sample or envelope hook in the instrument to
attach to. NULL, the default, means to use the default
hook, which depends on the type of slave Item.

For envelopes, the default envelope hook is "Env".

For samples, the default may only be used for master items
with one FIFO (either input or output). In this case, NULL
selects the one and only FIFO. If the master item instrument
has more than one FIFO, you must specify the name of the
FIFO to which to attach.

AF_TAG_SLAVE (Item) - Create, Query
Envelope or Sample Item to attach to Instrument.

AF_TAG_SET_FLAGS (uint32) - Create, Query, Modify
Set of AF_ATT_ flags to set. Sets every flag for which a 1
is set in ta_Arg.

AF_TAG_START_AT (int32) - Create, Query, Modify

Specifies the point at which to start when the attachment is started (with StartAttachment() or StartInstrument()).

For sample attachments, specifies a sample frame number in the sample at which to begin playback.

For envelopes attachments, specifies the segment index at which to start.

Flags

AF_ATTFFATLADYSINGS

If set, causes the instrument to stop when the attachment finishes playing. This flag can be used to mark the one or more Envelope(s) or Sample(s) that are considered to be the determiners of when the instrument is done playing and should be stopped.

For envelopes, the default setting for this flag comes from the AF_ENVF_FATLADYSINGS flag. Defaults to cleared for samples.

AF_ATTFFNOAUTOSTART

When set, causes StartInstrument() to not automatically start this attachment. This allows later starting of the attachment by using StartAttachment(). This is useful for sound spooling applications. Defaults to cleared (attachment defaults to starting when instrument is started).

See Also

Envelope, Instrument, Sample, Template

AudioClock

Audio virtual timer item.

Description

An AudioClock is a virtual timer that runs at a rate independent of other AudioClock. All AudioClocks are derived from the Codec Frame Clock which typically runs at 44100 Hz. An arbitrary number of independent AudioClocks may be created.

Most AudioClock functions accept an AudioClock Item or the constant AF_GLOBAL_CLOCK, which refers to a system-wide clock. This clock runs at a constant rate of approximately, but not exactly, 240 Hz.

Folio

audio

Item Type

AUDIO_CLOCK_NODE

Create

CreateAudioClock()

Delete

DeleteAudioClock()

Use

SignalAtAudioTime(), AbortTimerCue(), SetAudioClockDuration(),
SetAudioClockRate(), ReadAudioClock()

See Also

Cue

Bitmap

An Item describing an image buffer in RAM.

Description

This Item represents a region of memory containing imagery of specific dimensions and format. It is used for rendering and display operations.

Tag Arguments

BMTAG_WIDTH (uint32)

Width of the Bitmap, in pixels. Note that the system may modify this value, depending on Tag settings below.

BMTAG_HEIGHT (uint32)

Height of the Bitmap, in pixels. Note that the system may modify this value, depending on Tag settings below.

BMTAG_TYPE (uint32)

Type of Bitmap desired. See below for a description of available Bitmap types.

BMTAG_CLIPWIDTH (uint32)

Horizontal width of clip region, in pixels.

BMTAG_CLIPHEIGHT (uint32)

Vertical height of clip region, in pixels.

BMTAG_XORIGIN (uint32)

Horizontal rendering and clip region offset, in pixels.

BMTAG_YORIGIN (uint32)

Vertical rendering and clip region offset, in pixels.

BMTAG_BUFFER (void *)

Pointer to beginning (upper-left pixel) of image buffer in RAM.

BMTAG_DISPLAYABLE (Boolean)

If true (non-zero), specifies that the Bitmap is intended for display. The system will perform more rigorous checks to assure the hardware can display the buffer described by the Tag arguments.

BMTAG_RENDERABLE (Boolean)

If true (non-zero), specifies that the Bitmap is intended to receive triangle engine rendering output. The system will perform more rigorous checks to assure the hardware can render into the buffer described by the Tag arguments.

BMTAG_MPEGABLE (Boolean)

If true (non-zero), specifies that the Bitmap is intended to receive the output of an MPEG decompression operation. The system will perform more rigorous checks to assure the MPEG hardware

can decompress into the buffer described by the Tag arguments.

BMTAG_BUMPDIMS (Boolean)

Depending on the intended use, the hardware requires the Bitmap to be of specific dimensions. If the supplied dimensions are unsupported by the hardware, and the argument to this Tag is true (non-zero), the system will bump the width and height to the next highest values that the hardware can handle (each dimension is considered independently). This assures that your Bitmap will be at least as large as you request. The updated dimensions will be written to the Bitmap Item.

BMTAG_COALIGN (Item)

While the hardware can operate on image- and Z-buffers arbitrarily aligned, greatest performance is realized when the image- and Z-buffers start on "alternate" 4K pages (e.g. the image-buffer starts on an even-numbered page and the Z-buffer on an odd-numbered page). If the argument to this Tag is a valid Bitmap Item, and that Bitmap has a buffer attached, the `bm_BufMemCareBits` and `bm_BufMemStateBits` fields will be set to indicate the state the bits in the buffer pointer (passed in with BMTAG_BUFFER) must have to achieve optimum alignment with the other Bitmap (the argument to this Tag). A block of memory satisfying these requirements may be procured from `AllocMemMasked()`. Co-alignment "phase" reported by the system through these fields varies depending on the Bitmaps being co-aligned. If both Bitmaps are Z-buffers, the buffers will be "in phase" (e.g. both buffers will start in even-numbered pages). If neither Bitmap is a Z-buffer, the buffers will also be in phase. If one is a Z-buffer and the other is not, then the buffers will be "out of phase" (e.g. one buffer will start in an odd-numbered page and the other in an even-numbered page). Read the folio chapter for a more detailed explanation.

BMTAG_RESETCLIP (Boolean)

If this Tag is true (non-zero), the clip region boundaries are reset to the full dimensions of the Bitmap.

The following Bitmap types are available:

BMTYPE_16

Buffer is 16 bits per pixel (15 RGB, 1 DSB).

BMTYPE_32

Buffer is 32 bits per pixel (24 RGB, 7 alpha, 1 DSB).

BMTYPE_16_ZBUFFER

Buffer is 16 bits per pixel, intended as a Z-buffer for rendering operations.

Folio

graphics

Item Type

GFX_BITMAP_NODE

Create

CreateItem()

Delete

DeleteItem()

Use

GfxSendCommandList(), CreateView(), PixelAddress()

Associated Files

<:graphics:bitmap.h>

See Also

CreateItem()

Cue

Audio asynchronous notification item.

Description

"Cue the organist."

This Item type is used to receive asynchronous notification of some event from the audio folio:

- sample playback completion
- envelope completion
- audio timer request completion
- trigger going off

Each Cue has a signal bit associated with it. Because signals are task-relative, Cues cannot be shared between multiple tasks.

Folio

audio

Item Type

AUDIO_CUE_NODE

Create

CreateCue(), CreateItem()

Delete

DeleteCue(), DeleteItem()

Query

GetCueSignal()

Use

AbortTimerCue(), ArmTrigger(), MonitorAttachment(),
SignalAtAudioTime(), SleepUntilAudioTime()

See Also

Attachment, AudioClock

Envelope

Audio envelope.

Description

An envelope is a time-variant function which can be used to control parameters of sounds that are to change over time (e.g., amplitude, frequency, filter characteristics, modulation amount, etc.). An Envelope Item is the envelope function description. In order to use an Envelope, it must be attached to an instrument with an envelope hook (e.g., `envelope.dsp`, which simply outputs the Envelope's function as a control signal). Custom patches may also use envelopes.

An Envelope's function is constructed from an array of `EnvelopeSegments` (see structure definition below). The initial value of the envelope function is `envsegs[0].envs_Value`. The function value then proceeds to `envsegs[1].envs_Value`, taking `envsegs[0].envs_Duration` seconds to do so. Barring any loops, the function proceeds from one `envs_Value` to the next in this manner until the final `envs_Value` in the array has been reached. The final `envs_Duration` value in the array is ignored.

The function of each segment when used with `envelope.dsp` is linear, but other instruments could interpolate between the `envs_Values` in other ways.

Folio

audio

Item Type

`AUDIO_ENVELOPE_NODE`

Create

`CreateEnvelope()`, `CreateItem()`

Delete

`DeleteEnvelope()`, `DeleteItem()`

Query

`GetAudioItemInfo()`

Modify

`SetAudioItemInfo()`

Use

`CreateAttachment()`

Tags

Data:

`AF_TAG_ADDRESS (const EnvelopeSegment *)` - Create, Query, Modify*
Pointer to array of `EnvelopeSegments` used to define the envelope.

The length of the array is specified with `AF_TAG_FRAMES`. This data must remain valid for the life of the Envelope or until a different address is set with `AF_TAG_ADDRESS`.

If the Envelope is created with { `AF_TAG_AUTO_FREE_DATA`, `TRUE` }, then this data will be freed automatically when the Envelope is deleted. Memory used with `AF_TAG_AUTO_FREE_DATA` must be allocated with `MEMTYPE_TRACKSIZE` set.

`AF_TAG_AUTO_FREE_DATA` (bool) - Create

Set to `TRUE` to cause data pointed to by `AF_TAG_ADDRESS` to be freed automatically when Envelope Item is deleted. If the Item isn't successfully created, the memory isn't freed.

The memory pointed to by `AF_TAG_ADDRESS` must be freeable by `FreeMem` (Address, `TRACKED_SIZE`).

`AF_TAG_FRAMES` (int32) - Create, Query, Modify*

Number of EnvelopeSegments in array pointed to by `AF_TAG_ADDRESS`.

`AF_TAG_TYPE` (uint8) - Create, Query, Modify

Determines the signal type of the envelope data (i.e., the units for `envs_Value`). Must be one of the `AF_SIGNAL_TYPE_*` defined in `<:audio:audio.h>`. Defaults to `AF_SIGNAL_TYPE_GENERIC_SIGNED` on creation.

* These tags cannot be modified for an Envelope created with { `AF_TAG_AUTO_FREE_DATA`, `TRUE` }.

Loops:

`AF_TAG_RELEASEBEGIN` (int32) - Create, Query, Modify

Index in EnvelopeSegment array for beginning of release loop. -1 indicates no loop, which is the default on creation. If not -1, must \leq the value set by `AF_TAG_RELEASEEND`. Must also be $<$ the number of segments.

`AF_TAG_RELEASEEND` (int32) - Create, Query, Modify

Index in EnvelopeSegment array for end of release loop. -1 indicates no loop, which is the default on creation. If not -1, must \geq the value set by `AF_TAG_RELEASEBEGIN`. Must also be $<$ the number of segments.

`AF_TAG_RELEASEJUMP` (int32) - Create, Query, Modify

Index in EnvelopeSegment array to jump to on release. When set, release causes escape from normal envelope processing to the specified index without disturbing the current output envelope value. From there, the envelope proceeds to the next EnvelopeSegment from the current value. -1 to disable, which is the default on creation. Must be $<$ the number of segments minus one.

`AF_TAG_RELEASETIME_FP` (float32) - Create, Query, Modify

The time in seconds used when looping from the end of the release loop back to the beginning. Defaults to 0.0 on creation.

AF_TAG_SUSTAINBEGIN (int32) - Create, Query, Modify
Index in EnvelopeSegment array for beginning of sustain loop. -1 indicates no loop, which is the default on creation. If not -1, <= the value set by AF_TAG_SUSTAINEND. Must also be < the number of segments.

AF_TAG_SUSTAINEND (int32) - Create, Query, Modify
Index in EnvelopeSegment array for end of sustain loop. -1 indicates no loop, which is the default on creation. If not -1, >= the value set by AF_TAG_SUSTAINBEGIN. Must also be < the number of segments.

AF_TAG_SUSTAINTIME_FP (float32) - Create, Query, Modify
The time in seconds used when looping from the end of the sustain loop back to the beginning. Defaults to 0.0 on creation.

Time Scaling:

These tags define how the Envelope responds to the StartInstrument() tag AF_TAG_PITCH.

AF_TAG_BASENOTE (uint8) - Create, Query, Modify
MIDI note number of pitch at which pitch-based time scale factor is 1.0.

AF_TAG_NOTESPEROCTAVE (int8) - Create, Query, Modify
Number of semitones at which pitch-based time scale doubles. A positive value makes the envelope times shorter as pitch increases; a negative value makes the envelope times longer as pitch increases. Zero (the default) disables pitch-based time scaling.

Misc:

AF_TAG_CLEAR_FLAGS (uint32) - Create, Modify
Set of AF_ENVF_ flags (see below) to clear. Clears every flag for which a 1 is set in ta_Arg.

AF_TAG_SET_FLAGS (uint32) - Create, Query, Modify
Set of AF_ENVF_ flags (see below) to set. Sets every flag for which a 1 is set in ta_Arg.

Flags

AF_ENVF_LOCKTIMESCALE
When set, causes the Time Scale for this envelope to ignore the use AF_TAG_TIME_SCALE_FP in StartInstrument() This is useful if you are using multiple envelopes in an instrument and want some to be time scaled, and some not to be. Envelopes used as complex LFOs are often not time scaled.

This flag does not affect pitch-based time scaling.

AF_ENVF_FATLADYSINGS

The state of this flag indicates the default setting for the AF_ATTFF_FATLADYSINGS Attachment flag whenever this Envelope is attached to an Instrument.

Data Format

```
typedef struct EnvelopeSegment
{
    float32 envs_Value;           // Starting value of this segment of
    the array.                   // The units of this depend on the
                                // setting for the Envelope Item.

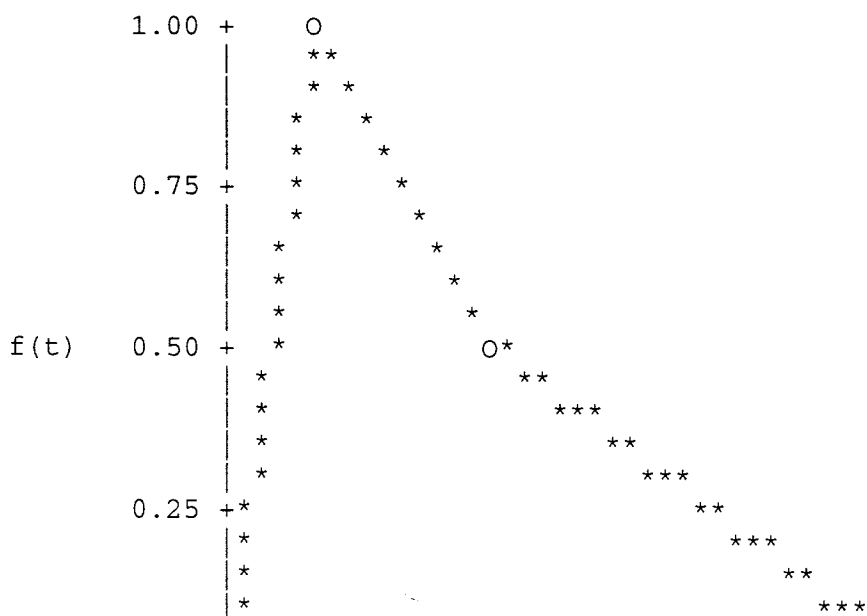
    float32 envs_Duration;        // Time in seconds to reach the
    envs_Value of                // the next EnvelopeSegment in the
    array.
} EnvelopeSegment;
```

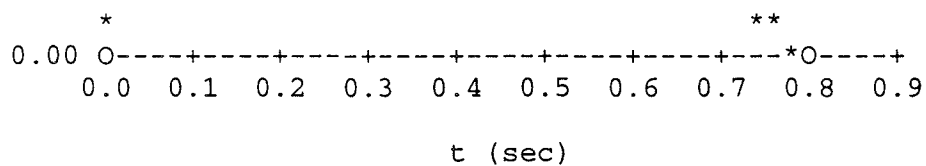
Example

The EnvelopeSegment array:

```
EnvelopeSegment envsegs[] = {
    // value, duration
    { 0.0, 0.1 },
    { 1.0, 0.2 },
    { 0.5, 0.5 },
    { 0.0, 0.0 }
};
```

corresponds to the graph:



**Caveats**

Care must be taken to avoid mismatching signed and unsigned Envelopes and Envelope instruments. A mismatch will result in a numeric overflow in the envelope instrument.

See Also

Attachment, Instrument, Template, `envelope.dsp`

ErrorText

A table of error messages.

Description

An error text item is a table of error messages. The kernel maintains a list of these items and uses them to convert a Portfolio error code into a descriptive string.

Folio

Kernel

Item Type

ERRORTXTNODE

Create

CreateItem()

Delete

DeleteItem()

Use

GetSysErr(), PrintfSysErr()

Tags

ERRTEXT_TAG_OBJID (uint32) - Create.

This tag specifies the 3 6-bit characters that identify these errors. You use the Make6Bit() macro to convert from ASCII into the 6-bit character set.

ERRTEXT_TAG_MAXERR (uint8) - Create.

Indicates the number of error strings being defined. This corresponds to the number of entries in the string table.

ERRTEXT_TAG_TABLE (const char **) - Create

A pointer to an array of string pointers. These strings will be used when converting an error code into a string. The array is indexed by the lower-8 bits of the error code.

File

A handle to a data file.

Description

A file item is created by the File folio when a file is opened. It is a private item used to maintain context information about the file being accessed.

Folio

file

Item Type

FILENODE

Create

`OpenFile()`, `ChangeDirectory()`

Delete

`CloseFile()`

Instrument

DSP Instrument Item.

Description

Instrument items are created from `Template` items. They correspond to actual instances of DSP code running on the DSP. Instrument ports can be connected together. Instruments can be played (by starting them), and controlled (by setting knobs). An Instrument is monophonic in the sense that it corresponds to a single voice (not mono vs stereo). Multiple voices require creating an Instrument per voice.

Folio

audio

Item Type`AUDIO_INSTRUMENT_NODE`**Create**`CreateInstrument(), CreateItem(), LoadInstrument()`**Delete**`DeleteInstrument(), DeleteItem(), UnloadInstrument()`**Query**`GetAudioItemInfo()`**Use**

`AbandonInstrument(), BendInstrumentPitch(), ConnectInstrumentParts(),
CreateAttachment(), CreateKnob(), CreateProbe(),
DisconnectInstrumentParts(), PauseInstrument(), ReleaseInstrument(),
ResumeInstrument(), StartInstrument(), StopInstrument(),
TuneInstrument()`

Tags

General:

AF_TAG_CALCRATE_DIVIDE (uint32) - Create

Specifies the denominator of the fraction of the total DSP cycles on which this instrument is to run. The valid settings for this are:

- 1 - Full rate execution (44,100 cycles/sec)
- 2 - Half rate (22,050 cycles/sec)
- 8 - 1/8 rate (5,512.5 cycles/sec) (new for M2)

Defaults to 1 on creation.

AF_TAG_PRIORITY (uint8) - Create, Query

The priority of execution in DSP in the range of 0..200, where 200 is the highest priority. Defaults to 100 on creation.

AF_TAG_SET_FLAGS (uint32) - Create

AF_INSF_ flags to set at creation time. Defaults to all cleared.

AF_TAG_START_TIME (AudioTime) - Query

Returns the AudioTime value of when the instrument was last started.

AF_TAG_STATUS (uint32) - Query

Returns the current instrument status: AF_STARTED, AF_RELEASED, AF_STOPPED, or AF_ABANDONED.

AF_TAG_TEMPLATE (Item) - Create

DSP Template Item used from which to create instrument.

Amplitude:

These tags apply to instruments that have an unconnected Amplitude knob. They are mutually exclusive except for AF_TAG_EXP_VELOCITY_SCALAR. All MIDI velocity tags also perform multi-sample selection based on Sample velocity ranges.

AF_TAG_AMPLITUDE_FP (float32) - Start

Value to set instrument's Amplitude knob to before starting instrument (for instruments that have an Amplitude knob). Valid range -1.0..1.0.

AF_TAG_VELOCITY (uint8) - Start

Linear MIDI key velocity to amplitude mapping:

$$\text{Amplitude} = \text{Velocity} / 127.0$$

MIDI key velocity is specified in the range of 0..127.

AF_TAG_SQUARE_VELOCITY (uint8) - Start

Like AF_TAG_VELOCITY except:

$$\text{Amplitude} = (\text{Velocity}/127.0)**2.$$

This gives a more natural response curve than the linear version.

AF_TAG_EXPONENTIAL_VELOCITY (uint8) - Start

Like AF_TAG_VELOCITY except:

$$\text{Amplitude} = 2.0**((\text{Velocity}-127)*\text{expVelocityScalar}).$$

where expVelocityScalar is set using AF_TAG_EXP_VELOCITY_SCALAR. This gives a more natural response curve than the linear version.

AF_TAG_EXP_VELOCITY_SCALAR (float32) - Start

Sets expVelocityScalar used by AF_TAG_EXPONENTIAL_VELOCITY. These two tags are used together with StartInstrument() and can be in either order. The default value is (1.0/20.0), which means that the Amplitude will double for every 20 units of velocity.

Frequency:

These tags apply to instruments that support frequency settings (e.g., oscillators, LFOs, sample players). They have no effect if the Frequency or SampleRate knob is connected to another instrument.

They are mutually exclusive.

AF_TAG_FREQUENCY_FP (float32) - Start

Play instrument at a specific output frequency.

Oscillator or LFO: Sets the Frequency knob to the specified value.

Sample player with a tuned sample: Adjusts the SampleRate knob to play the sample at the desired frequency. For example, to play sinewave.aiff at 440Hz, set this tag to 440.

AF_TAG_PITCH (uint8) - Start

MIDI note number of the pitch to play instrument at. The range is 0 to 127; 60 is middle C. Once the output frequency is determined applies the frequency in the same manner as AF_TAG_FREQUENCY_FP.

This tag also performs multi-sample selection based on Sample note ranges, and Envelope pitch-based time scaling.

AF_TAG_RATE_FP (float32) - Start

Set raw frequency control. For samplers, this is a fraction of DAC rate. For oscillators, this is a fractional phase increment.

AF_TAG_SAMPLE_RATE_FP (float32) - Start

Sample rate in Hz to set sample player's SampleRate knob to.

AF_TAG_DETUNE_FP (float32) - Start

For samplers, play at a fraction of original recorded sample rate. For oscillators or LFOs, play at a fraction of default frequency.

Time Scaling:

AF_TAG_TIME_SCALE_FP (float32) - Start, Release

Scales times for all Envelopes attached to this Instrument which do not have the AF_ENVF_LOCKTIMESCALE flag set. Defaults to 1.0 when instrument is started. Defaults to start setting when released.

Flags

AF_INSF_AUTOABANDON

When set, causes instrument to automatically go to the AF_ABANDONED state when stopped either automatically because of an AF_ATTFFATLADYSINGS flag, or manually because of a StopInstrument() call). Otherwise, the instrument goes to the AF_STOPPED state when stopped. Note that regardless of the state of this flag, instruments are created in the AF_STOPPED state.

See Also

Template, Attachment, Knob, Probe

IOReq

An item used to communicate between a task and an I/O device.

Description

An IOReq item is used to carry information from a client task to a device in order to have the device perform some operation. Once the device stores return information in the IOReq, and returns it to the client task.

Folio

Kernel

Item Type

IOREQNODE

Create

CreateIOReq(), CreateItem()

Delete

DeleteIOReq(), DeleteItem()

Query

FindItem()

Modify

SetItemOwner(), SetItemPri()

Use

SendIO(), DoIO(), AbortIO(), WaitIO(), CheckIO()

Tags

CREATEIOREQ_TAG_REPLYPORT (Item) - Create

The item number of a message port. This is where the device will send a message whenever an I/O operation involving this IOReq completes. If you do not specify this tag, the device will send your task the SIGF_iodone signal instead. This tag is mutually exclusive with the CREATEIOREQ_TAG_SIGNAL tag.

CREATEIOREQ_TAG_SIGNAL (int32) - Create

Specifies the signal mask to send to your task whenever an I/O operation involving this IOReq completes. If this tag is not supplied, the standard SIGF_iodone signal is sent instead. This tag is mutually exclusive with the CREATEIOREQ_TAG_REPLYPORT tag.

CREATEIOREQ_TAG_DEVICE (Item) - Create

This specifies the item number of the device that this IOReq will be used to communicate with. This item number is obtained by calling OpenDeviceStack().

Knob

An item for adjusting an Instrument's parameters.

Description

A Knob is an Item for adjusting an Instrument's parameters. Each knob has one or more parts (as defined by the instrument), all of which can be addressed from its knob item. Each knob has a signal type which defines the units of the knob's value. Knob have a default signal type as defined by the instrument, but this can be overridden by using `AF_TAG_TYPE` when creating the knob. Knobs have a default value per part, which can be read using `ReadKnobPart()` before that part has been set with `SetKnobPart()`.

Multiple knob items for the same instrument knob can coexist without conflict, even if they have different signal types. The last value set by `SetKnobPart()` takes precedence.

Folio

audio

Item Type

AUDIO_KNOB_NODE

Create

`CreateKnob()`

Delete

`DeleteKnob()`

Query

`GetAudioItemInfo()`

Use

`SetKnobPart()`, `ReadKnobPart()`

Tags

`AF_TAG_INSTRUMENT` (Item) - Create
Specifies instrument containing knob to create.

`AF_TAG_MAX_FP` (float32) - Query
Returns maximum value of knob in the units appropriate for the signal type specified by `AF_TAG_TYPE`, and taking into consideration execution rate.

`AF_TAG_MIN_FP` (float32) - Query
Returns minimum value of knob in the units appropriate for the signal type specified by `AF_TAG_TYPE`, and taking into consideration execution rate.

`AF_TAG_NAME` (const char *) - Create, Query
Knob name. On creation, specifies name of knob belonging to the instrument to create. On query, returns a pointer to knob's name.

`AF_TAG_TYPE` (uint8) - Create
Determines the signal type of the knob. Must be one of the `AF_SIGNAL_TYPE_*` defined in `<audio/audio.h>`. Defaults to the default signal type of the knob (for standard DSP instruments, this is described in the instrument documentation).

See Also

Instrument, Probe, GetInstrumentPortInfoByName(), GetAudioSignalInfo()

Locale

A database of international information.

Description

A Locale structure contains a number of definitions to enable software to operate transparently across a wide range of cultural and language environments.

Folio

International

Item Type

LOCALENODE

Use

```
intlOpenLocale(), intlCloseLocale(), intlLookupLocale()
```

Message

The means of sending data from one task to another.

Description

Messages are used to communicate amongst tasks. Messages come in three flavors: small, standard, and buffered. The flavors each carry their own type or amount of information. The small message carries exactly eight bytes of data. Standard messages contain a pointer to some data. Buffered messages contain an arbitrary amount of data within them.

Messages are closely associated with message ports. In order to send a message, a task must indicate a destination message port. Essentially, messages shuttle back and forth between message ports, and tasks can get and put messages from and to message ports.

Folio

Kernel

Item Type

MESSAGENODE

Create

`CreateMsg()`, `CreateSmallMsg()`, `CreateBufferedMsg()`, `CreateItem()`

Delete

`DeleteMsg()`, `DeleteItem()`

Query

`FindItem()`, `FindNamedItem()`

Use

`ReplyMsg()`, `ReplySmallMsg()`, `SendMsg()`, `SendSmallMsg()`, `GetThisMsg()`, `GetMsg()`, `WaitPort()`

Tags

`CREATMSG_TAG_REPLYPORT` (Item) - Create

The item number of the message port to use when replying this message. This must be a port that was created by the same task or thread that is creating the message. If you do not supply this tag, then it will not be possible to reply this message. Therefore, the act of sending a message with no reply port also transfers the ownership of the message to the receiving task.

`CREATMSG_TAG_MSG_IS_SMALL` (void) - Create

When this tag is present, it specifies that this message should be small. This means that this message should be used with `SendSmallMsg()` and `ReplySmallMsg()`, and can only pass 8 bytes of information.

`CREATMSG_TAG_DATA_SIZE` (uint32) - Create

When this tag is present, it specifies that the message should be buffered. The argument of this tag specifies the size of the buffer that should be allocated.

`CREATMSG_TAG_SIGNAL` (int32) - Create

Whenever a message arrives at a message port, the kernel sends a signal to the owner of the message port. This tag lets you specify which signal should be sent. If this tag is not provided,

the kernel will automatically allocate a signal for the message port.

CREATMSG_TAG_USERDATA (void *) - Create

Lets you specify the 32-bit value that gets put into the msg_UserData field of the Message structure. This can be anything you want, and is sometimes useful to identify a message.

MsgPort

An item through which a task receives messages.

Description

A message port is an item through which a task receives messages.

Folio

Kernel

Item Type

MSGPORTNODE

Create

`CreateMsgPort(), CreateUniqueMsgPort(), CreateItem()`

Delete

`DeleteMsgPort(), DeleteItem()`

Query

`FindMsgPort(), FindItem(), FindNamedItem()`

Use

`WaitPort(), GetMsg(), SendMsg(), SendSmallMsg(), ReplyMsg(), ReplySmallMsg()`

Tags

CREATEPORT_TAG_SIGNAL (int32) - Create

Whenever a message arrives at a message port, the kernel sends a signal to the owner of the message port. This tag lets you specify which signal should be sent. If this tag is not provided, the kernel will automatically allocate a signal for the message port.

CREATEPORT_TAG_USERDATA (void *) - Create

Lets you specify the 32-bit value that gets put into the `mp_UserData` field of the `MsgPort` structure. This can be anything you want, and is sometimes useful to identify a message port between tasks.

Probe

An item to permit the CPU to read the output of a DSP Instrument.

Description

A probe is an item that allows the CPU to read the output of a DSP instrument. It is the inverse of a Knob. Multiple probes can be connected to a single output. A probe does not interfere with a connection between instruments.

Folio

audio

Item Type

AUDIO_PROBE_NODE

Create

CreateProbe(), CreateItem()

Delete

DeleteProbe(), DeleteItem()

Use

ReadProbePart()

Tags

AF_TAG_TYPE (uint8) - Create

Determines the signal type of the probe. Must be one of the AF_SIGNAL_TYPE_* defined in <:audio:audio.h>. Defaults to the signal type of the output (for standard DSP instruments, this is described in the instrument documentation).

See Also

Instrument, Knob

Projector

An Item representing a particular physical display type.

Description

Projectors are the foundation for hardware-specific display manipulation; a form of video "device driver."

Projectors represent an abstract "display mode," a physical method of presenting imagery on a display. For example, the NTSC and PAL video standards are two different methods of displaying imagery, each having different characteristics such as refresh rate, maximum pixel dimensions, and so on. Likewise, NTSC interlaced and NTSC non-interlaced have important differences, such as number of fields per frame, and maximum number of displayable lines.

Projectors represent an abstract description of such characteristics. It's also an "anchor point" for View heirarchies. Indeed, each Projector can be thought of as the root ViewList for each supported display type.

Projectors are created and maintained by the graphics folio. Upon startup, the folio looks for, loads, and initializes every Projector in the system that can be supported by the underlying hardware.

More than one Projector can be associated with a given physical display. In such cases, only one Projector may be active at a given time. The View heirarchy attached to the active Projector is the one visible. View heirarchies attached to inactive Projectors are not visible.

Views and ViewLists are attached to Projectors using `AddViewToViewList()`, supplying the Item number of the Projector as the target ViewList.

Internal System Specifics

Projectors are implemented using the Portfolio device driver facilities, but do not actually employ the I/O model. The device driver facilities were used to leverage off the demand loading mechanism and DDF capabilities (allowing us to theoretically put any number of drivers on the disc, but only those that would actually work would get loaded). However, the I/O model was not used due to concerns about the overhead. Communication between the folio and the Projectors is accomplished through a set of function pointers in the Projector Item.

Tag Arguments

`PROJTAG_WIDTH (uint32)`

The maximum displayable width of the display, in display (View positioning) coordinates.

`PROJTAG_HEIGHT (uint32)`

The maximum displayable height of the display, in display (View positioning) coordinates.

`PROJTAG_PIXELWIDTH (uint32)`

The maximum displayable width of the display, in the smallest pixels available from this Projector.

`PROJTAG_PIXELHEIGHT (uint32)`

The maximum displayable height of the display, in the smallest pixels available from this Projector.

`PROJTAG_FIELDSPERSECOND (float32)`

The exact number of fields scanned out to the monitor each second. This is a float so it can

describe refresh rates that aren't exact integral values (like NTSC's, which is 59.94).

PROJTAG_FIELDSPERFRAME (uint8)

The number of fields required to describe a complete frame of imagery. NTSC is two. VGA-style displays are typically one.

PROJTAG_XASPECT (uint8)

The horizontal component of the aspect ratio of this Projector's squarest available pixels.

PROJTAG_YASPECT (uint8)

The vertical component of the aspect ratio of this Projector's squarest available pixels.

PROJTAG_PROJTYPE (int32)

An encoded Projector ID number. Although Projector IDs are stored as uint8s, they are presented to this Tag as pre-formatted int32s (as if they were View type identifiers).

PROJTAG_BLANKVIEWTYPE (ViewTypeInfo *)

A pointer to the ViewTypeInfo structure that should be used to compose blank Views. This is used by the folio's compiler service (SuperInternalCompile()) to build Transition segments for Views that are malformed or have no Bitmap attached.

PROJTAG_VEC_INIT (Err (*) ())

Pointer to Projector initialization vector. This is called as part of the CreateItem() sequence. "Global" driver and/or hardware stuff should be done in the driver's initialization routine. This vector is called to fill out the Projector Item structure itself.

The init vector is prototyped as follows:

```
Err init (struct Projector *p, struct GraphicsFolioBase *gb);
```

'p' is a pointer to the Projector Item you are expected to initialize. 'gb' is a pointer to graphics' Folio Item structure, in case there are any bits you need out of there.

PROJTAG_VEC_EXPUNGE (Err (*) ())

Pointer to the Projector expunge vector. This is called as part of the DeleteItem() sequence. "Global" driver teardown should not be done here; this is to properly clean up the Projector Item itself.

The expunge vector is prototyped as follows:

```
Err expunge (struct Projector *p);
```

'p' is a pointer to the Projector about to be deleted.

PROJTAG_VEC_SCAVENGE (Err (*) ()) [CURRENTLY UNIMPLEMENTED]

Pointer to the Projector's scavenge vector. This vector is intended to be called when the system runs low on memory, and needs components to clean up whatever they can. This vector should FreeMem() transient data structures where possible.

PROJTAG_VEC_LAMP (Err (*) ())

Pointer to the Projector's lamp vector. This vector is responsible for doing whatever tricks are necessary to make this Projector and its subordinate Views visible on the display. This may possibly entail shutting down another Projector using the same video hardware.

The lamp vector is prototyped as follows:

```
Err lamp (struct Projector *p, int32 op);
```

'p' is a pointer to the Projector being "lamped." 'op' is an operation code; zero means turn it off, non-zero means turn it on. If this was a request to turn the Projector on, and another Projector had to be turned off to accomplish it, the Item number of the turned-off Projector is returned.

```
PROJTAG_VEC_UPDATE (Err (*) ( ))
```

Pointer to the Projector's update vector. This vector is called in response to a variety of operations performed within the graphics folio. It gets called when a View is modified, moved, depth arranged, made visible, removed, or deleted.

The update vector is prototyped as follows:

```
Err update (struct Projector *p, int32 reason, void *ob);
```

Operation of the update vector is discussed in a later section.

```
PROJTAG_VEC_NEXTVIEWTYPE (ViewTypeInfo *(*) ( ))
```

Pointer to the Projector's nextviewtype vector. This is called by the folio when it is searching for a particular ViewTypeInfo structure.

The nextviewtype vector is prototyped as follows:

```
struct ViewTypeInfo *nextviewtype (struct Projector  
*p, struct ViewTypeInfo *vti);
```

'p' points to the Projector whose ViewTypeInfo database is being searched. 'vti' points to a ViewTypeInfo structure previously retrieved from the vector; if 'vti' is NULL, a pointer to the first available ViewTypeInfo structure is returned.

The vector returns a pointer to the next available ViewTypeInfo structure. If there are no ViewTypeInfos left for this Projector, NULL is returned.

Update Vector

The update vector is the most heavily trafficked routine in the Projector. It is the central contact point between the Projector and the core graphics folio for manipulating the display.

The update vector receives three parameters: a pointer to the Projector affected, an operation code, and a pointer to some pertinent data. The way the data pointer is interpreted changes based on the operation being requested.

The available operation codes are:

```
PROJOP_NOP
```

Do nothing; return immediately. 'ob' is undefined.

```
PROJOP_MODIFY
```

You are being called as part of a ModifyGraphicsItem() or a CreateItem() operation. 'ob' points to a ModifyInfo structure, which contains a pointer to the View ready to be integrated into the display, a pointer to the same View *before* it was modified (useful for certain optimizations), and a pointer to the base of the TagArg array that

was used to modify the View (also useful for optimizations).

PROJOP_ADDVIEW

A View is being added somewhere in the Projector's View heirarchy. 'ob' points to the View being added.

PROJOP_REMOVEVIEW

A View is being removed from somewhere in the Projector's View heirarchy. 'ob' points to the View being removed.

PROJOP_ORDERVIEW

A View is being depth-arranged somewhere in the Projector's View heirarchy. 'ob' points to the View being depth-arranged.

PROJOP_UNLOCKDISPLAY

The Projector has just been unlocked with `UnlockDisplay()`; the Projector should now integrate all outstanding changes into the visible display. 'ob' is undefined.

When called, the update vector should interpret the operation code and pertinent data and perform whatever operations are necessary to make the display an accurate reflection of the current state of the View heirarchy.

Upon successful completion, the vector should return zero, and the display should be up to date. If anything inside the update vector fails, the routine should leave the display unchanged and return a negative error code. This error code will be passed directly back to the application.

The vector is responsible for checking the `p_ViewListSema4` to see if it has been locked. If so, the display should *not* be updated, though the routine should still return a success code. The vector is free to update internal state; if such an internal update fails, a (negative) error code should be returned.

Caveats

I have not yet worked out how to specify `PROJTAG_WIDTH` and `PROJTAG_HEIGHT` for display/video hardware where Views cannot be positioned (like on NickNack, for instance).

Folio

graphics

Item Type

`GFX_PROJECTOR_NODE`

Create

`CreateItem()`

Delete

`DeleteItem()`

Use

`AddViewToViewList()`, `LockDisplay()`, `UnlockDisplay()`,
`SuperInternalCompile()`

Associated Files

`<:graphics:projector.h>`

See Also

CreateItem(), AddViewToViewList(), LockDisplay(), UnlockDisplay(),
SuperInternalCompile()

Sample

A digital recording of a sound.

Description

A Sample Item is a handle to a digital recording of a sound in memory. Samples come in two kinds:

Ordinary Samples

Samples that are playable by various instruments (e.g., `sampler_16_v1.dsp`). Memory for these samples is provided by the client.

Delay Lines

Special samples suitable for receiving the DMA output from delay instruments (e.g., `delay_f1.dsp`). The memory for delay lines is allocated from supervisor memory by the audio folio. Use `CreateDelayLine()` to create a delay line.

Most sample operations can be performed on both kinds of samples.

Folio

audio

Item Type

AUDIO_SAMPLE_NODE

Create

`CreateDelayLine()`, `CreateItem()`, `CreateSample()`

Delete

`DeleteDelayLine()`, `DeleteItem()`, `DeleteSample()`

Query

`GetAudioItemInfo()`

Modify

`SetAudioItemInfo()`

Use

`CreateAttachment()`, `DebugSample()`

Tags

Data:

AF_TAG_ADDRESS (const void *) - Create*, Query, Modify*+

Address of sample data. The length of the data may be specified in bytes using `AF_TAG_NUMBYTES` or in sample frames using `AF_TAG_FRAMES`. This data must remain valid for the life of the Sample or until another address is set with `AF_TAG_ADDRESS`.

Can set to NULL for a sample Item without data. This is useful if you intend to hook multiple data pointers to the Sample Item using `SetAudioItemInfo()` some time after creation. NULL address should be used with 0 length.

If the sample is created with { `AF_TAG_AUTO_FREE_DATA`, `TRUE` }, then this data will

be freed automatically when the Sample is deleted. Memory used with AF_TAG_AUTO_FREE_DATA must be allocated with MEMTYPE_TRACKSIZE set. NULL address and 0 size are not legal in this case.

Defaults to NULL on creation.

AF_TAG_AUTO_FREE_DATA (bool) - Create*

Set to TRUE to cause data pointed to by AF_TAG_ADDRESS to be automatically freed when Sample Item is deleted. If the Item isn't successfully created, the memory isn't freed.

The memory pointed to by AF_TAG_ADDRESS must be freeable by FreeMem (Address, TRACKED_SIZE).

AF_TAG_FRAMES (int32) - Create*, Query, Modify*+

Length of sample data expressed in frames. In a stereo sample, this would be the number of stereo pairs. Defaults to 0 on creation.

AF_TAG_NUMBYTES (int32) - Create*, Query, Modify*+

Length of sample data expressed in bytes. Defaults to 0 on creation.

AF_TAG_DELAY_LINE (int32) - Create

Number of bytes to allocate for delay line. Mutually exclusive with AF_TAG_ADDRESS, AF_TAG_FRAMES, AF_TAG_NUMBYTES, and AF_TAG_AUTO_FREE_DATA.

* These operations marked with '*' are allowed for ordinary samples, but not for delay lines.

+ These tags cannot be used to modify a Sample created with { AF_TAG_AUTO_FREE_DATA, TRUE }.

Format:

AF_TAG_CHANNELS (uint8) - Create, Query, Modify

Number of channels (or samples per sample frame). For example: 1 for mono, 2 for stereo. Valid range is 1..255. Defaults to 1 on creation.

AF_TAG_WIDTH (uint8) - Create, Query, Modify

Number of bytes per sample (uncompressed). Valid range is 1..2. Defaults to 2 on creation.

AF_TAG_NUMBITS (uint8) - Create, Query, Modify

Number of bits per sample (uncompressed). Valid range is 1..16. Width is rounded up to the next byte when computed from this tag. Defaults to 16 on creation.

AF_TAG_COMPRESSIONTYPE (PackedID) - Create, Query, Modify

32-bit ID representing AIFC compression type of sample data (e.g., ID_SDX2). Use 0 for no compression, which is the default for creation.

AF_TAG_COMPRESSIONRATIO (uint8) - Create, Query, Modify

Compression ratio of sample data. Uncompressed data has a value of 1. Valid range is 1..255. Defaults to 1 on creation.

Note: These tags affect the frame size and therefore also affect the relationship between AF_TAG_FRAMES

and AF_TAG_NUMBYTES.

Loops:

AF_TAG_SUSTAINBEGIN (int32) - Create, Query, Modify.

Frame index of the first frame of the sustain loop. Valid range is 0..NumFrames-1. -1 for no sustain loop. Use in conjunction with AF_TAG_SUSTAINEND.

AF_TAG_SUSTAINEND (int32) - Create, Query, Modify.

Frame index of the first frame after the last frame in the sustain loop. Valid range is 1..NumFrames. -1 for no sustain loop. Use in conjunction with AF_TAG_SUSTAINBEGIN.

AF_TAG_RELEASEBEGIN (int32) - Create, Query, Modify.

Frame index of the first frame of the release loop. Valid range is 0..NumFrames-1. -1 for no release loop. Use in conjunction with AF_TAG_RELEASEEND.

AF_TAG_RELEASEEND (int32) - Create, Query, Modify

Frame index of the first frame after the last frame in the release loop. Valid range is 1..NumFrames. -1 for no release loop. Use in conjunction with AF_TAG_RELEASEBEGIN.

Tuning:

AF_TAG_BASENOTE (uint8) - Create, Query, Modify

MIDI note number for this sample when played at the original sample rate (as set by AF_TAG_SAMPLE_RATE_FP). This defines the frequency conversion reference note for the StartInstrument() AF_TAG_PITCH tag. Defaults to middle C (60) on creation.

AF_TAG_DETUNE (int8) - Create, Query, Modify

Amount to detune the MIDI base note in cents to reach the original pitch. Must be in the range of -100 to 100.

AF_TAG_SAMPLE_RATE_FP (float32) - Create, Query, Modify

Original sample rate in Hz. Defaults to 44,100 Hz on creation.

AF_TAG_BASEFREQ_FP (float32) - Query

The frequency of the sample, in Hz, when played at the DAC sample rate (as returned by GetAudioFolioInfo()) using the AF_TAG_SAMPLE_RATE_FP tag). This value is computed from the other tuning tag values.

Multisample:

AF_TAG_LOWNOTE (uint8) - Create, Query, Modify

Lowest MIDI note number at which to play this sample when part of a multisample. StartInstrument() AF_TAG_PITCH tag is used to perform selection. Valid range is 0 to 127. Defaults to 0 on creation.

AF_TAG_HIGHNOTE (uint8) - Create, Query, Modify

Highest MIDI note number at which to play this sample when part of a multisample. Valid range is 0 to 127. Defaults to 127 on creation.

AF_TAG_LOWVELOCITY (uint8) - Create, Query, Modify

Lowest MIDI attack velocity at which to play this sample when part of a multisample. `StartInstrument()` velocity tags (e.g., `AF_TAG_VELOCITY`) are used to perform selection. Range is 0 to 127. Defaults to 0 on creation.

`AF_TAG_HIGHVELOCITY` (uint8) - Create, Query, Modify

Highest MIDI attack velocity at which to play this sample when part of a multisample. Range is 0 to 127. Defaults to 127 on creation.

Caveats

All sample data, loop points, and lengths must be byte aligned. For example, a 1-channel, ADPCM sample (which has 4 bits per frame) is only legal when the lengths and loop points are at multiples of 2 frames.

In the development version of the audio folio, frame- and byte-aligned sample lengths and byte-aligned loop points are enforced. In the production version of the audio folio, these traps are removed with the expectation that you are using valid lengths and loop points. Failure to do so may result in noise at loop points, or slight popping at the ending of sound playback. It is recommended that you pay very close attention to sample lengths and loop points when creating, converting, and compressing samples.

See Also

`Attachment`, `Instrument`, `Template`, `CreateSample()`, `StartInstrument()`, `sampler_16_v1.dsp`, `delay_f1.dsp`

Semaphore

An item used to arbitrate access to shared resources.

Description

Semaphores are used to protect shared resources. Before accessing a shared resource, a task must first try to lock the associated semaphore. Only one task at a time can have the semaphore locked at any one moment. This prevents multiple tasks from accessing the same data at the same time.

Folio

Kernel

Item Type

SEMA4NODE

Create

CreateSemaphore(), CreateUniqueSemaphore()

Delete

DeleteSemaphore(),

Query

FindSemaphore()

Use

LockSemaphore(), UnlockSemaphore()

Tags

CREATESEMAPHORE_TAG_USERDATA (void *) - Create

Lets you specify the 32-bit value that gets put into the sem_UserData field of the Semaphore structure. This can be anything you want, and is sometimes useful to identify a semaphore port between tasks.

Task

An executable context.

Description

A task item contains all the information needed by the kernel to support multitasking. It contains room to store CPU registers when the associated task context goes to sleep, it contains pointers to various resources used by the task, and it specifies the task's priority. There is one task item for every task or thread that exists.

Folio

kernel

Item Type

TASKNODE

Create Call`CreateThread(), CreateTask()`**Delete**`DeleteThread(), DeleteItem()`**Query**`FindTask()`**Modify**`SetItemOwner(), SetItemPri()`**Tags****TAG_ITEM_NAME** (const char *) - Create

Lets you specify the name of the task. This is a required tag.

TAG_ITEM_PRI (uint8) - Create

Provide a priority for the task in the range 10 to 199. If this tag is not given, the task gets it either from the supplied module item, or inherits it from the creator.

CREATETASK_TAG_MODULE (Item) - Create

This tag specifies the module item where the code and data for this task can be found.

CREATETASK_TAG_PC (void *) - Create

Provide a pointer to the code to be executed.

CREATETASK_TAG_SP (void *) - Create

Provide a pointer to the memory buffer to use as stack for a thread. This tag is only valid when starting a thread. This pointer must be aligned on an 8 byte boundary, and points to one byte past the end of the allocated stack area.

CREATETASK_TAG_STACKSIZE (uint32) - Create

Specifies the size in bytes of the memory buffer reserved for a thread's stack. This must be a multiple of 8 bytes.

CREATETASK_TAG_ARGC (uint32) - Create

A 32-bit value that will be passed to the task or thread being launched as its first argument.

If this is omitted, the first argument will be 0.

CREATETASK_TAG_ARGP (void *) - Create

A 32-bit value that will be passed to the task or thread being launched as a second argument. If this is omitted, the second argument will be 0.

CREATETASK_TAG_MSGFROMCHILD (Item) - Create

Provides the item number of a message port. The kernel will send a status message to this port whenever the thread or task being created exits. The message is sent by the kernel after the task has been deleted. The msg_Result field of the message contains the exit status of the task. This is the value the task provided to `exit()`, or the value returned by the task's `main()` function. The msg_DataPtr field of the message contains the item number of the task that just terminated. Finally, the msg_DataSize field contains the item number of the thread or task that terminated the task. If the task exited on its own, this will be the item number of the task itself. It is the responsibility of the task that receives the status message to delete it when it is no longer needed by using `DeleteMsg()`.

CREATETASK_TAG_FREESTACK (void) - Create

When this tag is supplied, it tells the kernel to automatically free the memory used for the stack when the thread dies. If the tag is not provided, the creator is responsible to free the stack. When using this tag, the memory cannot have been allocated with `MEMTYPE_TRACKSIZE`.

CREATETASK_TAG_MAXQ (uint32) - Create

A value indicating the maximum quanta for the task in microseconds.

CREATETASK_TAG_USERDATA (void *) - Create

This specifies an arbitrary 32-bit value that is put in the new task's `t_UserData` field. This is a convenient way to pass a pointer to a shared data structure when starting a thread.

CREATETASK_TAG_USEREXCHANDLER (UserHandler) - Create

Lets you specify the exception handler for the task.

CREATETASK_TAG_DEFAULTMSGPORT (void) - Create

When this tag is present, the kernel automatically creates a message port for the new task or thread being started. The item number of this port is stored in the Task structure's `t_DefaultMsgPort` field. This is a convenient way to quickly establish a communication channel between a parent and a child.

CREATETASK_TAG_THREAD (void)

This tag specifies that you wish to start a thread instead of a full-fledged task.

TEContext

An Item describing the context of the Triangle Engine.

Description

This Item acts as a storage space for the current state of the triangle engine. The state of the triangle engine consists of 1) the contents of the triangle engine hardware registers, 2) the contents of the texture RAM, 3) the contents of the PIPRAM.

Any combination of the above elements may be specified to be saved when the triangle engine is "context switched" (a distinct operation from task switching).

TEContext Items are attached to I/O requests submitted to the triangle engine device. If a given I/O request has a TEContext attached, it is checked to see if it the same as the last TEContext received by the device. If they are the same, no special action is taken. If they are different, the triangle engine elements requested by the previous TEContext are stored within the Item. The requested elements stored within the new TEContext are then loaded into the triangle engine.

I/O requests submitted without an attached TEContext will likewise trigger a store into any previous TEContext.

Creating a TEContext Item, in addition to memory for the Item, will also allocate system RAM to store the specified triangle engine elements; storage will not be allocated for elements not requested.

A TEContext Item may be attached to any number of I/O requests.

Tag Arguments

TECONTEXT_TAG_FLAGS (uint32)

A bitfield specifying the triangle engine elements to be saved/restored.

The flags which may be specified for TECONTEXT_TAG_FLAGS are:

TECF_SAVE_REGISTERS

Save/restore the triangle engine registers.

TECF_SAVE_TRAM

Save/restore the texture RAM.

TECF_SAVE_PIPRAM

Save/restore the PIPRAM.

Folio

graphics

Item Type

GFX_TECONTEXT_NODE

Create

CreateItem()

Delete

DeleteItem()

Associated Files

`<:graphics:graphics.h>`

See Also

`CreateItem()`, `DeleteItem()`, `CreateTEIOReq()`, `DeleteTEIOReq()`,
`GFXCMD_EXECUTEITEMS`

Template

A description of an audio instrument.

Description

A Template is the description of a DSP audio instrument (including the DSP code, resource requirements, parameter settings, etc.) from which Instrument items are created. A Template can either be a standard system instrument template (e.g. `envelope.dsp`, `sampler_16_v1.dsp`, etc), a mixer template, or a custom patch template (e.g. created with ARIA).

Folio

audio

Item Type

AUDIO_TEMPLATE_NODE

Create`CreateMixerTemplate()`, `CreatePatchTemplate()`, `LoadInsTemplate()`**Delete**`DeleteItem()`, `DeleteMixerTemplate()`, `DeletePatchTemplate()`,
`UnloadInsTemplate()`**Use**`CreateInstrument()`, `AdoptInstrument()`, `CreateAttachment()`,
`TuneInsTemplate()`**See Also**

Instrument, Attachment, Sample

Tuning

An item that contains information for tuning an Instrument.

Description

A tuning item contains information for converting MIDI pitch numbers to frequency for an Instrument or Template. The information includes an array of frequencies, and 32-bit integer values indicating the number of pitches, notes in an octave, and the lowest pitch for the tuning frequency. See the `CreateTuning()` call in the "Audio Folio Calls" chapter in this manual for more information.

Folio

audio

Item Type

AUDIO_TUNING_NODE

Create

`CreateItem()`, `CreateTuning()`

Delete

`DeleteItem()`, `DeleteTuning()`

Modify

`SetAudioItemInfo()`

Use

`TuneInsTemplate()`, `TuneInstrument()`

Tags

AF_TAG_ADDRESS (const float32 *) - Create, Modify*

Array of frequencies in float32 Hz to be used as a lookup table. The length of the array is specified with AF_TAG_FREAMES. This data must remain valid for the life of the Tuning or until a different address is set with AF_TAG_ADDRESS.

If the Tuning Item is created with { AF_TAG_AUTO_FREE_DATA, TRUE }, then this data will be freed automatically when the Tuning is deleted. Memory used with AF_TAG_AUTO_FREE_DATA must be allocated with MEMTYPE_TRACKSIZE set.

AF_TAG_AUTO_FREE_DATA (bool) - Create

Set to TRUE to cause data pointed to by AF_TAG_ADDRESS to be freed automatically when Tuning Item is deleted. If the Item isn't successfully created, the memory isn't freed.

The memory pointed to by AF_TAG_ADDRESS must be freeable by FreeMem (Address, TRACKED_SIZE).

AF_TAG_BASENOTE (uint8) - Create, Modify

MIDI note number that should be given the first frequency in the frequency array in the range of 0..127.

AF_TAG_FRAMES (int32) - Create, Modify*

Number of frequencies in array pointed to by AF_TAG_ADDRESS. This value must be \geq NotesPerOctave.

AF_TAG_NOTESPEROCTAVE (int32) - Create, Modify

Number of notes per octave. This is used to determine where in the frequency array to look for notes that fall outside the range of BaseNote..BaseNote+Frames-1. This value must be \leq Frames.

* These tags cannot be modified for a Tuning created with { AF_TAG_AUTO_FREE_DATA, TRUE }.

See Also

Instrument, Template

View

An Item describing a displayed region of imagery.

Description

This Item represents an image displayed on the physical monitor. It contains information describing its position, colors, depth arrangement, display modes, and other characteristics.

Tag Arguments

VIEWTAG_VIEWTYPE (int32)

Specifies the type of View to be constructed. See "View Types" below for a list of currently available types.

VIEWTAG_WIDTH (uint32)

Specifies the width of the View, in display coordinates.

VIEWTAG_HEIGHT (uint32)

Specifies the height of the View, in display coordinates.

VIEWTAG_TOPEDGE (uint32)

Specifies where the top edge of the View is to be located on the display, in display coordinates.

VIEWTAG_LEFTEDGE (uint32)

Specifies where the left edge of the View is to be located on the display, in display coordinates.

VIEWTAG_WINTOPEDGE (uint32)

Specifies the topmost pixel from the Bitmap which is to be the first pixel displayed at the top edge of the View, in pixels.

VIEWTAG_WINLEFTEDGE (uint32)

Specifies the leftmost pixel from the Bitmap which is to be the first pixel displayed at the left edge of the View, in pixels.

VIEWTAG_BITMAP (Item)

The Item number of the Bitmap to be displayed through this View.

VIEWTAG_PIXELWIDTH (uint32)

Specifies the width of the View, in pixels of the underlying Bitmap.

VIEWTAG_PIXELHEIGHT (uint32)

Specifies the height of the View, in pixels of the underlying Bitmap.

VIEWTAG_AVGMODE (int32)

Specifies whether Opera-style horizontal and/or vertical averaging is to be enabled within this View. See "Averaging Modes" below for a list of available modes.

VIEWTAG_RENDER SIGNAL (uint32)

A bitmask specifying the signals to be asserted when the Bitmap underlying the View may be safely rendered into. If zero, no signal is to be asserted.

VIEWTAG_DISPLAY SIGNAL (uint32)

A bitmask specifying the signals to be asserted when the Bitmap underlying the View has been seen by the user at least once. If zero, no signal is to be asserted.

VIEWTAG_SIGNALTASK (Item)

The Item number of the task to be signalled when the Bitmap may be safely rendered, and when the Bitmap has been seen by the user. If NULL (which is the default), the owner of the Item is signalled.

VIEWTAG_BESILENT (Boolean)

Signals are sent for all modifications to a View, including its creation. This can be inconvenient at times. If this Tag is present and its argument is true (non-zero), no notification signal will be asserted for the modification in which this Tag was present. This feature applies only to specific modifications, not the Item itself. Modifications not specifying this Tag will continue to generate signals.

The following is an incomplete list of currently available View types. Note that this scheme of View typing/naming is under review, and is going to change. Watch for it. (See the include file `<:graphics:view.h>` for a complete list of available View types.)

VIEWTYPE_16

View is 16 bits per pixel, 320 * 240 nominal resolution.

VIEWTYPE_16_LACE

View is 16 bits per pixel, 320 * 480 nominal resolution.

VIEWTYPE_16_640

View is 16 bits per pixel, 640 * 240 nominal resolution.

VIEWTYPE_16_640_LACE

View is 16 bits per pixel, 640 * 480 nominal resolution.

VIEWTYPE_32

View is 32 bits per pixel, 320 * 240 nominal resolution.

VIEWTYPE_32_LACE

View is 32 bits per pixel, 320 * 480 nominal resolution.

VIEWTYPE_32_640

View is 32 bits per pixel, 640 * 240 nominal resolution.

VIEWTYPE_32_640_LACE

View is 32 bits per pixel, 640 * 480 nominal resolution.

The following averaging modes are available. The AVGMODEs are bitfields, and may be set independently of each other:

AVGMODE_H

Horizontal interpolation enabled.

AVGMODE_V

Vertical interpolation enabled.

AVG_DSB_0 AVG_DSB_1

Constant values to OR with the above AVGMODE values. If AVG_DSB_0 is OR'd, the specified averaging mode is applied to pixels whose DSB bit is zero. Similarly, OR'ing AVG_DSB_1

will apply the specified averaging mode to pixels whose DSB bit is set to one.

Caveats

Bitmap types and View types cannot be freely intermixed. That is, for a Bitmap to be displayed, it must be handed to a View of a compatible type. You can't, for example, display a `BMTYPE_16` through a `VIEWTYPE_32`. However, handing a `BMTYPE_16` to a `VIEWTYPE_16_LACE` is perfectly cool. (This mechanic may change.)

The `VIEWTYPE_` names are all going to change to something more reasonable. Watch for it. (No, really, they will. Seriously. I'm not kidding...)

`VIEWTAG_{WIDTH,HEIGHT}` and `VIEWTAG_PIXEL{WIDTH,HEIGHT}` are mutually exclusive. That is, setting `_PIXELWIDTH` will undo any previous setting to `_WIDTH`, and vice-versa.

Opera-style averaging may be unavailable for certain View types. No error will be reported; it just won't do anything.

For BDA 1.1, enabling vertical averaging may cause cosmetic boo-boos where Views overlap.

Setting signalling on a View to signal the Item's owner (the default) and then changing the owner of the View Item is currently undefined. Don't do dat.

The nominal pixel resolutions given for the above View types are for NTSC mode. PAL dimensions are different. Inspect the `ViewTypeInfo` structure pointed to by the View's `v_ViewTypeInfo` field to discover the View type's full pixel dimensions.

Folio

graphics

Item Type

`GFX_VIEW_NODE`

Create

`CreateItem()`

Delete

`DeleteItem()`

Use

`AddViewToViewList()`, `OrderViews()`, `RemoveView()`

Associated Files

`<:graphics:view.h>`

See Also

`CreateItem()`, `Bitmap`, `ViewList`

ViewList

An anchor for a heirarchy of sub-Views.

Description

This Item is used as an anchor point for a heirarchy of subordinate View and ViewList Items. With a ViewList, an entire heirarchy of Views may be moved or depth-arranged as a single unit. This permits the application -- or the user -- to move an application's Views without disturbing their relative positions and layering priority.

ViewLists do not themselves display any imagery.

Tag Arguments

VIEWTAG_LEFTEDGE (int32)

Position of the left-most edge of the ViewList, in display coordinates. This value will be used to offset all subordinate View and ViewList Items.

VIEWTAG_TOPEDGE (int32)

Position of the top-most edge of the ViewList, in display coordinates. This value will be used to offset all subordinate View and ViewList Items.

Note

You are strongly urged to give your root ViewList Item a human-readable name (using TAG_ITEM_NAME) so that they may be meaningfully interpreted and manipulated by the user.

Folio

graphics

Item Type

GFX_VIEWLIST_NODE

Create

CreateItem()

Delete

DeleteItem()

Use

AddViewToViewList(), OrderViews(), RemoveView()

Associated Files

<:graphics:view.h>

See Also

CreateItem(), AddViewToViewList(), OrderViews(), RemoveView(), View

Chapter 16

Requester Folio Calls

This section presents the reference documentation for the Requester folio, which provides file management user-interface components.

CreateStorageReq

Create a storage requester object.

Synopsis

```
Err CreateStorageReq(StorageReq **req, const TagArg *tags);
```

```
Err CreateStorageReqVA(StorageReq **req, uint32 tags, ...);
```

Description

This function creates a storage requester object. This object can be used to display a user-interface to allow the user to interact with the file system.

Arguments

req

A pointer to a StorageReq variable, where a handle to the requester object can be stored.

tags

A pointer to an array of tag arguments containing data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

STORREQ_TAG_VIEW_LIST (Item)

This tag lets you specify the view list into which the requester being created should be displayed. If this tag is not supplied, the requester will be displayed in the primary view list.

STORREQ_TAG_FONT (Item)

This specifies the font to use for textual displays. If this is not supplied, a default font is used. Supplied fonts should have a 2:1 aspect ratio, since the display used is 640x240 pixels.

STORREQ_TAG_LOCALE (Item)

This specifies the locale to use for language and for number formatting. If this tag is not supplied, the current system locale is used.

STORREQ_TAG_OPTIONS (uint32)

This tag supplies a bitmask of options that control whether various features are available to the user. See the many STORREQ_OPTION_XXX constants in *<:ui:requester.h>* for details.

STORREQ_TAG_PROMPT (SpriteObj *)

This specifies the sprite to display as a prompt to the user. If this tag is not supplied, no prompt is displayed.

STORREQ_TAG_FILTERFUNC (FileFilterFunc)

This lets you specify a function pointer that is used to determine whether or not individual files or directories should be displayed to the user. The function gets called for every file that is scanned, and is supplied the directory name and a DirectoryEntry structure describing the file of interest. If the function returns TRUE, then the file is displayed to the user. If this tag is not supplied, the default is to display all files.

STORREQ_TAG_DIRECTORY (const char *)

This lets you specify the initial directory to display when bringing up the storage requester. This can be an absolute path (something starting with /) or a path relative to the current directory. If this tag is not supplied, or if the supplied directory can't be found, then the current directory will be used. If the current directory is not available, then another suitable directory will be displayed instead.

STORREQ_TAG_FILE (const char *)

This lets you specify the initial file to select within the displayed directory. If this is not supplied, or if that file doesn't exist within the directory, a suitable default will be selected.

Return Value

Returns ≥ 0 if successful, or a negative error code upon failure.

Implementation

Folio call implemented in requester folio V30.

Associated Files

<:ui:requester.h>, System.m2/Modules/requester

See Also

DisplayStorageReq(), DeleteStorageReq(), ModifyStorageReq(),
QueryStorageReq()

DeleteStorageReq

Delete a storage requester object.

Synopsis

```
Err DeleteStorageReq(StorageReq *req);
```

Description

This function deletes a storage requester object. This frees up any resources allocated when the object was created. Once the object is deleted, it can no longer be used.

Arguments

req

The storage requester object to delete. This may be NULL in which case this function does nothing.

Return Value

Returns ≥ 0 if successful, or a negative error code upon failure.

Implementation

Folio call implemented in requester folio V30.

Associated Files

<:ui:requester.h>, System.m2/Modules/requester

See Also

DisplayStorageReq(), CreateStorageReq(), ModifyStorageReq(),
QueryStorageReq()

DisplayStorageReq

Display a storage requester object to the user.

Synopsis

```
Err DisplayStorageReq(StorageReq *req);
```

Description

This function displays a storage requester to the user. The user will be able to navigate the file system and make selections. The function only returns when the user is done interacting with the object.

Warning

Once this calls returns, the state of the graphics hardware is indeterminate. You should not assume anything about register and TRAM contents.

Arguments

req
The storage requester object to display to the user.

Return Value

STORREQ_CANCEL

The user chose to cancel the operation. The app should return to the same state it was before the storage requester was displayed.

STORREQ_OK

The user made a choice and wishes the app to proceed with the operation.

A negative error code is returned upon failure.

Implementation

Folio call implemented in requester folio V30.

Associated Files

<:ui:requester.h>, System.m2/Modules/requester

See Also

CreateStorageReq(), DeleteStorageReq(), ModifyStorageReq(),
QueryStorageReq()

ModifyStorageReq

Modify attributes of a storage requester object.

Synopsis

```
Err ModifyStorageReq(StorageReq *req, const TagArg *tags);
```

```
Err ModifyStorageReqVA(StorageReq *req, uint32 tags, ...);
```

Description

This function lets you modify attributes of an existing storage requester object. The object attributes are persistent throughout the life of the object. Using this function you can change any individual attribute. Any attribute not specified by the tag list will not be affected.

Arguments

req

A pointer to the storage requester object to affect.

tags

A pointer to an array of tag arguments containing data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

STORREQ_TAG_VIEW_LIST (Item)

This tag lets you specify the view list into which the requester being created should be displayed.

STORREQ_TAG_FONT (Item)

This specifies the font to use for textual displays. Supplied fonts should have a 2:1 aspect ratio, since the display used is 640x240 pixels.

STORREQ_TAG_LOCALE (Item)

This specifies the locale to use for language and for number formatting.

STORREQ_TAG_OPTIONS (uint32)

This tag supplies a bitmask of options that control whether various features are available to the user. See the many STORREQ_OPTION_XXX constants in `<:ui:requester.h>` for details.

STORREQ_TAG_PROMPT (SpriteObj *)

This specifies the sprite to display as a prompt to the user.

STORREQ_TAG_FILTERFUNC (FileFilterFunc)

This lets you specify a function pointer that is used to determine whether or not individual files or directories should be displayed to the user. The function gets called for every file that is scanned, and is supplied the directory name and a DirectoryEntry structure describing the file of interest. If the function returns TRUE, then the file is displayed to the user.

STORREQ_TAG_DIRECTORY (const char *)

This lets you specify the initial directory to display when bringing up the storage requester.

This can be an absolute path (something starting with /) or a path relative to the current directory.

STORREQ_TAG_FILE (const char *)

This lets you specify the initial file to select within the displayed directory.

Return Value

>= 0 if successful, or a negative error code upon failure.

Implementation

Folio call implemented in requester folio V30.

Associated Files

<:ui:requester.h>, System.m2/Modules/requester

See Also

DisplayStorageReq(), DeleteStorageReq(), CreateStorageReq(),
QueryStorageReq()

QueryStorageReq

Query the current state of certain attributes of a storage requester object.

Synopsis

```
Err QueryStorageReq(StorageReq *req, const TagArg *tags);
```

```
Err QueryStorageReqVA(StorageReq *req, uint32 tag, ...);
```

Description

This function lets you query the current state of certain attributes of a storage requester object. You provide a tag list of the attributes to query. The parameter for each tag points to a memory location where the resulting information should be stored.

Arguments

- req**
A pointer to the requester object to query.
- tags**
A pointer to an array of tag arguments containing data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

STORREQ_TAG_VIEW_LIST (Item *)

This queries the requester's view list attribute. This will return -1 if no explicit view list was given.

STORREQ_TAG_FONT (Item *)

This queries the font that the requester is using for textual displays.

STORREQ_TAG_LOCALE (Item *)

This queries the locale that the requester is using for formatting numbers and language selection.

STORREQ_TAG_OPTIONS (uint32)

This returns the set of options currently active for this requester. This is a bitmask of the STORREQ_OPTION_XXX constants from *<ui/requester.h>*

STORREQ_TAG_PROMPT (SpriteObj **)

This returns a pointer to the SpriteObj used to prompt the user.

STORREQ_TAG_FILTERFUNC (FileFilterFunc *)

This returns a pointer to the current file filtering function, or NULL if no filtering function is currently installed.

STORREQ_TAG_DIRECTORY (char *)

This tag specifies a pointer to a string buffer. QueryStorageReq() will copy the name of the current directory into this buffer. The size of the buffer must be at least FILESYSTEM_MAX_NAME_LEN bytes.

STORREQ_TAG_FILE (char *)

This tag specifies a pointer to a string buffer. `QueryStorageReq()` will copy the name of the currently selected file into this buffer. The size of the buffer must be at least `FILESYSTEM_MAX_NAME_LEN` bytes.

Return Value

`>= 0` if successful, or a negative error code upon failure.

Implementation

Folio call implemented in requester folio V30.

Associated Files

`<:ui:requester.h>`, `System.m2/Modules/requester`

See Also

`CreateStorageReq()`, `DeleteStorageReq()`, `ModifyStorageReq()`,
`DisplayStorageReq()`

Chapter 17

SaveGame Folio Calls

This section presents the reference documentation for the SaveGame folio, which provides simple methods for loading and saving compressed saved game files to storage.

LoadGameData

Loads a saved game file into memory.

Synopsis

```
Err LoadGameData(LoadedGame *game, const TagArg *tags);  
Err LoadGameDataVA(LoadedGame *game, uint32 tag1, ... );
```

Description

This function scans a standard saved-game file, and attempts to load the application specific section into the application defined buffer.

Arguments

game

A pointer to a LoadedGame structure that will be filled in with associated data upon successful completion.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

LOADGAMEDATA_TAG_FILENAME

An absolute or relative pathname for the saved game file. Mutually exclusive with LOADGAMEDATA_TAG_IFFPARSER.

LOADGAMEDATA_TAG_IFFPARSER

An IFFParser * to read game data from. Mutually exclusive with LOADGAMEDATA_TAG_FILENAME.

LOADGAMEDATA_TAG_BUFFERARRAY

An array of SaveGameData structures, each containing a pointer to buffer memory, and a length value for that particular buffer. An entry with a length value of 0 terminates the array. This array is used to load in individual data chunks as they're encountered. The folio will fill in the Actual field of each structure with the amount of data it actually stored there. Mutually exclusive with the tag LOADGAMEDATA_TAG_CALLBACK.

LOADGAMEDATA_TAG_CALLBACK (GSCallBack *)

A pointer to a callback function that is called each time the folio encounters a new chunk of saved game data. When a new chunk of saved game data is encountered, the folio fills in the Length field of a SaveGameData structure with the size of the data that it would like to load, then passes that structure (along with any app-private data defined by LOADGAME_DATA_TAG_CALLBACKDATA) to the callback routine. At this point, the app has several options. If it would like to load the data, it should provide a buffer pointer in the Buffer field of the SaveGameData structure and return. If the app would like to skip the data, it should modify the Length field to 0 and return. If the app ever returns a negative number from the callback, the LoadSaveGame() function will fail, interpreting that as an error. This tag is mutually exclusive with the tag LOADGAMEDATA_TAG_BUFFERARRAY.

LOADGAMEDATA_TAG_CALLBACKDATA (void *)

This tag, valid only when presented with the tag `LOADGAMEDATA_TAG_CALLBACK`, defines application-private data to be provided each time the callback function is used.

`LOADGAMEDATA_TAG_ICON` (Icon **)

When provided, this tag requests that any Icon encountered in the game file be loaded and a pointer to the Icon structure be stored in the given pointer. If no Icon is encountered, the pointer will return as `NULL`. Note that if an Icon is loaded, it is your responsibility to free it (with the `UnloadIcon()` function in the icon folio).

`LOADGAMEDATA_TAG_IDSTRING` (char *)

When provided, this tag requests that the game file's ID string be copied into this location. Any string buffer provided should be a minimum of 32 characters in length.

Return Value

If positive, an indication of success. If negative, a system-standard error code.

Implementation

Folio call implemented in `SaveGame` folio V30.

Associated Files

`<:misc:savegame.h>`

See Also

`SaveGameData()`, `SaveGameDataVA()`

SaveGameData

Stores a saved game file.

Synopsis

```
Err SaveGameData(const char *appname, const TagArg *tags);  
Err SaveGameDataVA(const char *appname, uint32 tag1, ... );
```

Description

This function saves a standard saved-game file.

Arguments

appname

A pointer to a string (maximum of 32 characters in length) that uniquely identifies the name of the application that created the save game file.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

SAVEGAMEDATA_TAG_FILENAME (const char *)

An absolute or relative pathname for the saved game file. Mutually exclusive with SAVEGAMEDATA_TAG_IFFPARSER.

SAVEGAMEDATA_TAG_IFFPARSER (IFFParser *)

An IFFParser * to save game data to. Mutually exclusive with SAVEGAMEDATA_TAG_FILENAME.

SAVEGAMEDATA_TAG_IDSTRING (char *)

A pointer to a string (maximum of 32 characters in length) that represents what this particular game is. [IE, 'Bob on Level 3'] If not provided, the string 'Saved Game' is used.

SAVEGAMEDATA_TAG_ICON (const char *)

A pathname to a UTF file (containing one or more textures) to be used as the icon imagery for this saved game file.

SAVEGAMEDATA_TAG_TIMEBETWEENFRAMES (TimeVal *)

The amount of time that a reader of this icon should be instructed to wait between displaying each frame of the icon. [Not necessary if only 0 or 1 frame is provided.]

SAVEGAMEDATA_TAG_BUFFERARRAY (SaveGameData *)

An array of SaveGameData structures, each containing a pointer to buffer memory and a length value for that particular buffer. An entry with a length value of 0 terminates the array. This array is used to save out particular chunks of data to the IFF file. Mutually exclusive with SAVEGAMEDATA_TAG_CALLBACK.

SAVEGAMEDATA_TAG_CALLBACK (GSCallBack *)

A pointer to a callback function that is called iteratively to locate the next chunk of game data to save. The function is called with a pointer to a SaveGameData struct that the function

is required to fill in, and an application-private value defined in SAVEGAMEDATA_TAG_CALLBACKDATA. When all data has been written, the app should fill in the length field of the SaveGameData structure with '0' to terminate the process. Mutually exclusive with SAVEGAMEDATA_TAG_BUFFERARRAY.

SAVEGAMEDATA_TAG_CALLBACKDATA (void *)

This tag, valid only when presented with the tag SAVEGAMEDATA_TAG_CALLBACK, defines application-private data to be provided each time the callback function is used.

Return Value

Positive or equal to zero indicates success. A negative number describes a system-standard error code.

Implementation

Folio call implemented in SaveGame folio V30.

Associated Files

<:misc:savegame.h>

See Also

LoadGameData(), LoadGameDataVA()

Chapter 18

Script Folio Calls

This section presents the reference documentation for the Script folio, which provides services to execute simple scripts.

CreateScriptContext

Creates and initializes a ScriptContext structure.

Synopsis

```
Err CreateScriptContext(ScriptContext **sc, const TagArg *tags);
```

```
Err CreateScriptContextVA(ScriptContext **sc, uint32 tag, ...);
```

Description

This function allocates and initializes a ScriptContext structure which is used to preserve state information across multiple calls to ExecuteCmdLine().

Arguments

sc

A pointer to a variable that will receive the ScriptContext pointer.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

SCRIPT_TAG_BACKGROUND_MODE (bool)

Sets the state of the background flag in the shell. When TRUE, programs being executed will run in the background.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Script folio V27.

Associated Files

<.:misc:script.h>, System.m2/Modules/script

See Also

DeleteScriptContext(), ExecuteCmdLine()

DeleteScriptContext

Deletes a ScriptContext structure.

Synopsis

```
Err DeleteScriptContext (ScriptContext *sc);
```

Description

This function releases any resources allocated by `CreateScriptContext()`.

Arguments

sc
A pointer to the ScriptContext structure, or NULL.

Return Value

Returns ≥ 0 for success, or a negative error code for failure.

Implementation

Folio call implemented in Script folio V27.

Associated Files

<:misc:script.h>, System.m2/Modules/script

See Also

`CreateScriptContext()`, `ExecuteCmdLine()`

ExecuteCmdLine

Executes a shell command-line.

Synopsis

```
Err ExecuteCmdLine(const char *cmdLine, int32 *pStatus, const TagArg  
*tags);
```

```
Err ExecuteCmdLineVA(const char *cmdLine, int32 *pStatus, uint32  
tag, ...);
```

Description

This function lets you execute a command-line as if it were typed in at a shell prompt.

Arguments

cmdLine

The command-line to execute. This can be anything you would type at a 3DO shell prompt.

pStatus

A pointer to an int32 into which is stored the exit status of the command-line when it terminates. This value is meaningful only if ExecuteCmdLine returns a value ≥ 0 , and if the command-line was executed in foreground mode.

tags

A pointer to an array of tag arguments containing extra data for this function. See below for a description of the tags supported.

Tag Arguments

The following tag arguments may be supplied in array form to this function. The array must be terminated with TAG_END.

SCRIPT_TAG_CONTEXT (ScriptContext *)

Specifies a script context structure. This is used to hold state information accross multiple calls to this function. If this tag is not supplied, then default values will be used.

SCRIPT_TAG_BACKGROUND_MODE (bool)

Sets the state of the background flag in the shell. When TRUE, programs being executed will run in the background.

Return Value

Returns ≥ 0 if the cmd-line was executed, or a negative error code for failure.

Implementation

Folio call implemented in Script folio V27.

Associated Files

<:misc:script.h>, System.m2/Modules/script

See Also

CreateScriptContext(), DeleteScriptContext(), system()

Chapter 19

Single Precision Math Library Calls

This section presents the reference documentation for the floating point math functions provided as part of the 3DO development environment.

ceilf

round towards positive infinity

Synopsis

```
float ceilf( float x )
```

Description

ceilf returns the smallest integral value not less than its input. In other words, it rounds towards positive infinity. point.

Arguments

x

input value

Return Value

A float equal to the smallest integral value not less than x.

Implementation

Assembly function in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

floorf(), fmodf()

fabsf

Floating Point Absolute Value

Synopsis

```
float fabsf( float x )
```

Description

This function returns the absolute value of a floating point number. Calling this functions is more efficient than using a macro or if statement.

Arguments

x
A floating point number

Return Value

Returns x or -x

Implementation

Inline assembly function in math.h and a link library function in libspmath.

Associated Files

<math.h>, libspmath.a

floorf

round towards negative infinity

Synopsis

```
float floorf( float x )
```

Description

floorf returns the largest integral value not greater than its input. In other words, it rounds towards negative infinity. point.

Arguments

x
input value

Return Value

A float equal to the largest integral value not greater than x.

Implementation

Assembly function in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

ceilf(), fmodf()

fmodf

get the remainder of a division

Synopsis

```
float fmodf( float x, float y )
```

Description

fmodf returns the floating point remainder of x/y.

Arguments

x,y
floating point numbers

Return Value

The floating point remainder of x/y. If y is 0.0, 0.0 is returned.

Implementation

C function in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

floorf(), fceilf()

modff

get integer and fractional parts of a float

Synopsis

```
float modff( float frac, float *iptr )
```

Description

modff splits a float into an integer and a fractional part. For example, modff(5.75, &i) will return 0.75 and set integer i to 5.

Arguments

frac

floating point number

iptr

pointer to where you want the integer part stored

Return Value

The fractional value is returned as a float.

Implementation

Assembly function in libspmath.

Associated Files

<math.h>, libspmath.a

expf

Exponential Function

Synopsis

```
float expf( float x )
```

Description

This function computes the exponential function of x .

Arguments

x
input value

Return Value

Returns e^x . If x is too large, the result will overflow, `errno` will be set to `ERANGE`, and `INF` will be returned. If x is too small, the result will underflow, `errno` will be set to `ERANGE`, and `0.0` will be returned.

Implementation

C function in `libspmath`.

Associated Files

`<math.h>`, `libspmath.a`

See Also

`logf()`

frexpf

get fractional and exponent parts of a float

Synopsis

```
float frexpf( float x, int *n )
```

Description

For an input float 'x', returns a float 'f' and integer 'n' such that $x = f * 2^n$.

Arguments

x	input float
n	pointer to where the integer part is stored

Return Value

If x is 0, returns 0. Otherwise, returns a float between .5 and 1.0.

Implementation

Assembly function in libspmath.

Caveats

Assembly programmers: Destroys the contents of registers r4-r5. Uses stack pointer (r1) for temporary storage.

Associated Files

<math.h>, libspmath.a

See Also

ldexpf(), modff()

ldexpf

multiply a float by a power of 2

Synopsis

```
float ldexpf( float num, int exp )
```

Description

For an input float 'num', and integer 'exp' returns a float 'x' such that $x = \text{num} * 2^{\text{exp}}$. Note that this is not faster than multiplying by a constant, but is much faster than using `pow()`. For example, using $x = 64.0 * y$ is faster than $x = \text{ldexpf}(y, 6)$. However, $x = \text{ldexpf}(y, z)$ is much faster than $x = y * \text{powf}(z, 2.0)$.

Arguments

num
input float

exp
power of 2 to multiply num by

Return Value

Returns a floating point number equal to $\text{num} * 2^{\text{exp}}$. An exception will occur on overflow.

Implementation

Assembly functions in libspmath.

Caveats

Assembly programmers: Destroys the contents of registers r4-r5. Uses stack pointer (r1) for temporary storage.

Associated Files

`<math.h>`, libspmath.a

See Also

`frexpf()`, `modff()`

log10f

Base-10 Logarithm

Synopsis

```
float log10f( float x )
```

Description

This function computes the base-10 logarithm of the input value.

Arguments

x
input float

Return Value

If x is less than or equal to 0, errno is set to EDOM and -INF is returned. Otherwise a float is returned.

Implementation

C function in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

logf(), powf()

logf

Natural Logarithm

Synopsis

```
float logf( float x )
```

Description

This function computes the natural logarithm of the input value.

Arguments

x
input float

Return Value

If x is less than or equal to 0, errno is set to EDOM and -INF is returned. Otherwise a float is returned.

Implementation

C function in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

log10f(), powf()

powf

Power Function

Synopsis

```
float powf( float x, float y )
```

Description

This function does exponentiation. It returns the first number raised to the second number.

Arguments

x

input float

y

float representing the power to raise x to.

Return Value

If x is equal to 0 and y is less than or equal to 0, or y is not integral, errno is set to EDOM and 0 is returned. Otherwise, x^y is returned.

Implementation

C function in libspmath.

Associated Files

`<math.h>`, libspmath.a

See Also

`logf()`, `log10f()`

**rsqrtf
rsqrtff.br rsqrtfff**

Reciprocal Square Root Functions

Synopsis

```
float rsqrtf( float x )  
float rsqrtff( float x )  
float rsqrtfff( float x )
```

Description

These functions return the reciprocal square root of the input value. `rsqrtf()` is as accurate as possible with single precision floating point. `rsqrtff()` is less accurate, but faster. `rsqrtfff()` is a quick approximation.

Function	Speed	Max Error	-----	-----	-----	rsqrtf	xx	0.00003% rsqrtff
xx	0.04% rsqrtfff		xx	1.7%				

Arguments

x
input value

Return Value

Returns $1.0/\sqrt{x}$. If x is less than or equal to 0, then an exception will occur.

Implementation

Assembly functions in libspmath. `rsqrtfff()` is also declared as an inline assembly function in `math.h`

Associated Files

`<math.h>`, `libspmath.a`

See Also

`sqrtf()`, `sqrtff()`, `sqrtfff()`

**sqrtf
sqrtff.br sqrtfff**

Square Root Functions

Synopsis

```
float sqrtf( float x )  
float sqrtff( float x )  
float sqrtfff( float x )
```

Description

These functions return the square root of the input value. `sqrtf()` is as accurate as possible with single precision floating point. `sqrtff()` is less accurate, but faster. `sqrtfff()` is a quick approximation.

Function	Speed	Max Error	-----	-----	-----	sqrtf	xx	0.00002%	sqrtff
xx	0.04%	sqrtfff		xx	1.7%				

Arguments

x
input value

Return Value

Returns the square root. If x is negative, then an exception will occur.

Implementation

Assembly functions in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

`rsqrtf()`, `rsqrtff()`, `rsqrtfff()`

**acosf
asinf.br atanf**

Inverse Trigonometric Functions

Synopsis

```
float acosf( float x )  
float asinf( float x )  
float atanf( float x )
```

Description

These functions return a number whose cosine/sine/tangent is x;

Arguments

x
input value

Return Value

Returns a floating point number whose cosine/sine/tangent is equal to x. For `acosf()` and `asinf()`, if the absolute value of x is greater than 1.0, `errno` will be set to `EDOM` and NaN will be returned.

Implementation

C functions in `libspmath`.

Associated Files

`<math.h>`, `libspmath.a`

See Also

`sinf()`, `cosf()`, `tanf()`

atan2farctangent of y/x **Synopsis**

```
float atan2f( float y, float x )
```

Description

This function computes the arctangent of y/x .

Arguments

x, y
input floating point values

Return Value

Returns the arctangent of y/x in the range of $[-\pi, +\pi]$. If x and y are both 0, returns 0.0.

Implementation

C function in libspmath.

Associated Files

`<math.h>`, libspmath.a

See Also

`atanf()`

cosf
sinf

Cosine and Sine Functions

Synopsis

```
float cosf( float x )  
float sinf( float x )
```

Description

These functions return the cosine or sine of the input value. These are as accurate as possible with single precision floating point.

Arguments

x
input in radians

Implementation

Assembly functions in libspmath.

Associated Files

<math.h>, libspmath.a

See Also

tanf(), asinf(), acosf()

**coshf
sinhf.br tanhf**

Hyperbolic Functions

Synopsis

```
float coshf( float x )  
float sinhf( float x )  
float tanhf( float x )
```

Description

These functions return the hyperbolic cosine/sine/tangent.

Arguments

x
input value in radians

Return Value

Returns a floating point number in the range of the function. If the return value is too large to be represented in a float, errno will be set to ERANGE and INF will be returned.

Implementation

C functions in libspmath.

Associated Files

<math.h>, libspmath.a



3DO M2 Supplemental Portfolio Reference

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

1

Example Programs

AutoMapper	Illustrates how to move and map the corners of a sprite	MPR-3
autosizeview	Illustrates how to create a Bitmap and View automatically sized for the prevailing video mode.	MPR-4
basicview	Illustrates how to create a basic View	MPR-5
compression	Demonstrates use of the compression folio.	MPR-6
DebugConsole	Demonstrates use of the debugging console	MPR- 7
defaultport	Demonstrates using a task's default message port to communicate with a child thread.	MPR-8
fasttiming	Demonstrates how to use the high accuracy kernel timing services	MPR-9
Ls	Displays the contents of a directory.	MPR-10
memdebug	Demonstrates the memory debugging subsystem.	MPR-11
metronome	Demonstrates how to use the metronome timer feature	MPR-12
msgpassing	Demonstrates sending and receiving messages between two threads.	MPR-13
numinfo	Example program used to demonstrate 3DODebug functionality.	MPR-14
signals	Demonstrates how to use signals	MPR-15
SpriteExample_1	Illustrates how to display a single sprite	MPR-16
timerread	Demonstrates how to use the timer device to read the current system time.	MPR-17
timersleep	Demonstrates how to use the timer device to wait for an amount of time specified on the command-line	MPR-18
Type	Types a file's content to the output terminal	MPR-19
Walker	Recursively displays the contents of a directory, and all nested directories.	MPR-20

Audio

auto_bea	Automatic rhythm demo that uses lots of AIFF library samples.	MPR-21
capture_audio	Record the output from the DSPP to a host file.	MPR-22
MarkovMusic	Constantly-varying environmentally-aware soundtrack.	MPR-23
minmax_audio	Measures the maximum and minimum output from the DSP.	MPR-25
playpimap	Listen to the instruments assigned in a pimap.....	MPR-26
playsample	Plays an AIFF sample in memory using the control pad.	MPR-27
sfx_score	Uses libmusic.a score player as a sound effects manager.	MPR-28
simple_envelope	Simple audio envelope example.....	MPR-30
ta_attach	Experiments with sample attachments.....	MPR-31
ta_customdelay.....	Demonstrates a delay line attachment.	MPR-32
ta_envelope	Tests various envelope options by passing test index.	MPR-33
ta_pitchnotes	Plays a sample at different MIDI pitches.....	MPR-34
ta_spool.....	Demonstrates the libmusic.a sound spooler.....	MPR-35
ta_sweeps.....	Demonstrates adjusting knobs.	MPR-36
ta_timer	Demonstrates use of the audio timer.	MPR-37
ta_tuning	Demonstrates custom tuning a DSP instrument.	MPR-38
tj_canon.....	Uses the juggler to create and play a semi-random canon.....	MPR-39
tj_multi	Uses the juggler to play a collection.	MPR-40
tj_simple.....	Uses the juggler to play two sequences.....	MPR-41
tone	Simple audio demonstration.	MPR-42
tsp_algorithmic.....	Advanced sound player example showing algorithmic sequencing of sound playback.....	MPR-43
tsp_rooms	Room-sensitive soundtrack example using advanced sound player.	MPR-45
tsp_spoolsoundfile	Plays an AIFF sound file from a thread using the advanced sound player..	MPR-47
tsp_switcher	Advanced sound player example that switches between sounds based on control pad input.	MPR-48
windpatch	Creates a "wind" sound effect using the audio folio's patch compiler.....	MPR-49

Beep

tb_envelope	Trigger envelopes using the Beep folio.	MPR-50
tb_playsamp	Play a sample using the Beep Folio.....	MPR-51
tb_spool.....	Spool audio from memory using the Beep folio.....	MPR-52

EventBroker

cpdump	Queries the event broker and prints out a summary of what's connected to the control port.....	MPR-53
focus.....	Talks to the event broker and switches the focus to a different listener	MPR-54
lookie	Connects to the event broker and reports any events that occur.	MPR-55

luckie	Uses the event broker to read events from the first control pad.	MPR-56
maus	Uses the event broker to read events from the first mouse.....	MPR-57

Frame2D

DrawLines	Illustrates three methods of drawing lines.	MPR-58
MoveSprite	Illustrates how to display, move, rotate and scale a single sprite.	MPR-59
Points	Illustrates how to draw points.	MPR-60
Rectangles	Illustrates how to move and map the corners of a sprite.	MPR-61
RenderOrder	Illustrates how to link sprites and control their rendering order.	MPR-62
SimpleSprite	Illustrates how to display a single sprite.	MPR-63
spin3d2d	Rotate 2d and 3d objects together	MPR-64

Framework

sceneperf	Measure the performance of scenes.	MPR-65
-----------------	---	--------

graphics

m2perf	Measure the raw performance of the M2	MPR-67
--------------	---	--------

Streaming

DataPlayer	A DataStream DATA subscriber example program.	MPR-73
EZFlixPlayer	A DataStream example program that plays synchronized video and audio.	MPR-74
PlaySA	A DataStream example program that plays streamed audio.....	MPR-75
VideoPlayer	A DataStream example program that plays synchronized video and audio.	MPR-77

2

Shell Commands

AcroAdmin	Acrobat Filesystem Administration Utility	MPR-81
AcroFormat	Format acrobat filesystems.....	MPR-82
Clock	Gets/sets the date and time from the battery-backed clock.....	MPR-83
Copy	Copies files or directories	MPR-84
Delete	Deletes files and directories.	MPR-85
Dismount	Dismounts a file system.	MPR-86
dumplogs	Dump event logs to the debugging terminal.	MPR-87
expunge	Remove unused demand-loaded modules from memory.	MPR-88
FileAttrs	Gets or sets attributes of a file.	MPR-89
FindFile	Searches for a file, and prints its pathname.	MPR-90
FSInfo	Displays information on mounted file systems.	MPR-91
HW	Prints a list of all hardware resources currently in the system.....	MPR-92
Intl	Gets or sets the international settings of the machine.	MPR-93

Items	Displays lists of active items	MPR-94
killtask	Remove an executing task or thread from the system.	MPR-95
log	Control event logging.....	MPR-96
Ls.....	Displays the contents of a directory.....	MPR-97
MinimizeFS	Minimize filesystem/folio memory footprint.....	MPR-98
MkDir.....	Creates new directories.....	MPR-99
Mount	Mounts a file system.....	MPR-100
MountLevel.....	Displays or changes the current filesystem automounter level	MPR-101
Options	Controls various system run-time options.....	MPR-102
ProxyFile.....	Proxy a file so that it becomes a block-oriented device	MPR-103
RecheckFS	Recheck all filesystems to see if they are still online.....	MPR-105
Rename	Renames a file or directory.....	MPR-106
RmDir	Removes directories.....	MPR-107
setalias.....	Set a file path alias.....	MPR-108
setbg.....	Set the shell's default behavior to background execution mode.	MPR-109
setcd	Set the shell's current directory.....	MPR-110
setfg.....	Set the shell's default behavior to foreground execution mode.	MPR-111
setmaxmem	Set the amount of memory available in the system to the maximum amount possible.....	MPR-112
setminmem.....	Set the amount of memory available in the system to the minimum amount of memory guaranteed to be available in a production environment.....	MPR-113
setpri	Set the shell's priority.....	MPR-114
showavailmem.....	Display information about the amount of memory currently available in the system.....	MPR-115
showcd	Show the name of the current directory.....	MPR-116
showerror	Display an error string associated with a system error code.	MPR-117
showfreeblocks.....	Show the contents of a memory list.....	MPR-118
showmemmap	Display a page map showing which pages of memory are used and free in the system, and which task owns which pages.....	MPR-119
showtask	Display information about tasks in the system.....	MPR-120
sleep	Cause the shell to pause for a number of seconds.....	MPR-121
SyncStress.....	Put some stress of task synchronization code to uncover title bugs.....	MPR-122
Type	Types a file's content to the output terminal.....	MPR-123
TypeIFF.....	Displays contents of an IFF file.....	MPR-124
Walker	Recursively displays the contents of a directory, and all nested directories.....	MPR-125
WriteMedia	Writes new raw data to media.....	MPR-126

Audio

audioavail.....	Show available audio resources in system.....	MPR-127
dspfaders.....	Try out DSP instruments or patches.....	MPR-128
insinfo.....	Displays information about DSP Instruments.	MPR-131
makepatch	Reads patch script language, writes binary patch file.	MPR-132
playmf.....	Plays a standard MIDI file.....	MPR-137

3

MPEG Video Decompression Device

Using MPEG in a 3DO Title	MPR-140
Benefits of MPEG	MPR-140
Typical Data Flow	MPR-140
Video Synchronization	MPR-141
Using the Portfolio MPEG Device	MPR-141
Common Operations	MPR-142
Video Decoding Options	MPR-151
Output Modes	MPR-151
Other Commands	MPR-153
MPEG Still Image Decoding.....	MPR-155
MPEG Device Driver Memory Allocation	MPR-155

Preface

About This Book

3DO M2 Supplemental Portfolio Reference contains miscellaneous example, system, and audio commands and call descriptions for the 3DO M2 Portfolio operating system.

About the Audience

This book is written for programmers and application developers. To use this document, you should have a working knowledge of the C programming language, and object-oriented concepts.

How This Book Is Organized

This manual is a command reference. It contains examples of M2 Portfolio procedure calls, syntax, and use. It contains the following chapters:

Chapter 1, Example Programs—Presents the reference documentation for the example programs included with Release 1.1 of the Portfolio operating system.

Chapter 2, Shell Commands—Lists the reference documentation for the shell commands used during development.

Related Documentation

The following manuals are useful to developers who are programming in the 3DO environment:

3DO M2 Portfolio Programmer's Guide. A guide to the features of the kernel, IO, filesystem, and so on in the 3DO operating system.

3DO M2 Programmer's Reference's. A guide to the function calls for the kernel, IO, and filesystem of the 3DO operating system.

3DO M2 Audio and Music Programmer's Guide. Describes the music and audio folios. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Graphics Programmer's Guide. Describes the function calls for the M2 graphics architecture, folios, and APIs.

3DO M2 Graphics Programmer's Reference. Describes the M2 graphics architecture and APIs. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Audio and Music Programmer's Guide. Describes the music and audio folios. It includes overviews, programming tutorials, sample code, and procedure call definitions.

3DO M2 Debugger Programmer's Guide. A guide to using the 3DO Debugger. It provides a tutorial on using the debugger as well as descriptions of the graphical user interface.

3DO M2 DataStreamer Programmer's Guide. A guide to the 3DO DataStreamer architecture, streaming tools, and weaver tool.

3DO M2 DataStreamer Programmer's Reference. Provides manual pages for the structures in the 3DO DataStreamer library, for all data preparation tools, and for all weaver script commands.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	<code>int32 OpenGraphicsFolio(void)</code>
procedure name	<code>CreateScreenGroup()</code>
new term or emphasis	A <i>ViewList</i> is a special case of a View.
file or folder name	The <i>remote</i> folder, the <i>demo.scr</i> file.

Chapter 1

Example Programs

This section presents the reference documentation for the example programs included with this release of Portfolio.

AutoMapper

Illustrates how to move and map the corners of a sprite.

Synopsis

AutoMapper <filename.utf>

Description

This program loads a given utf file into a sprite and allows the user to manipulate each corner of the sprite. If no control pad input is received after about five seconds, the program enters an automatic mode, moving the sprite around the screen.

Associated Files

AutoMapper.c

Location

TBD

autosizeview

Illustrates how to create a Bitmap and View automatically sized for the prevailing video mode.

Synopsis

`autosizeview`

Description

Based on `basicview`, this program demonstrates a more sophisticated method of Bitmap and View creation, adjusting automatically for the prevailing video mode.

The program creates a View, inspects the system-maintained structures describing the View characteristics, and builds a display based on that information. The result is a Bitmap and View which cover the display, no matter which video mode (NTSC, PAL, etc.) is active.

Associated Files

`autosizeview.c`

Location

`examples/graphics/graphicsfolio`

See Also

`basicview`

basicview

Illustrates how to create a basic View.

Synopsis`basicview`**Description**

This program demonstrates the creation of a Bitmap, its buffer, and a View. The program places the View on the display, fills it with blue, sleeps for about five seconds, then tears everything down and exits.

For the View creation, this example leverages off the system default View settings, based on the Bitmap supplied. These defaults may be escaped by supplying additional TagArgs specifying the desired characteristics.

Associated Files`basicview.c`**Location**`examples/graphics/graphicsfolio`**See Also**`autosizeview`

compression

Demonstrates use of the compression folio.

Synopsis

compression

Description

Simple program demonstrating how to use the compression routines supplied by compression folio. The program loads itself into a memory buffer, compresses the data, decompresses it, and compares the original data with the decompressed data to make sure the compression and decompression processes worked successfully.

Associated Files

compression.c

Location

examples/Miscellaneous/Compression

DebugConsole

Demonstrates use of the debugging console.

Synopsis

debugconsole

Description

Simple program demonstrating how to use the debugging console, which provides a simple and efficient way to perform somewhat real-time debugging output.

Associated Files

debugconsole.c

Location

examples/Miscellaneous/DebugConsole

defaultport

Demonstrates using a task's default message port to communicate with a child thread.

Synopsis

defaultport

Description

Demonstrates how to start a thread having a default message port, and use the port for communication.

Associated Files

defaultport.c

Location

examples/Kernel

fasttiming

Demonstrates how to use the high accuracy kernel timing services.

Synopsis

fasttiming

Description

This program demonstrates how to use the high accuracy low overhead timing services provided by the kernel.

Note that this particular example will only give a relatively accurate reading of how much time it takes to call `Yield()` if the shell is running in foreground mode. Otherwise, the shell task will interfere with the execution of this program, and will therefore skew the results.

Associated Files

fasttiming.c

Location

examples/Kernel

Ls

Displays the contents of a directory.

Synopsis

```
Ls {directory}
```

Description

Demonstrates how to scan a directory to determine its contents.

Arguments

{directory}

Name of the directories to list. You can specify an arbitrary number of directory names. If no name is specified, the directory in which the ls program is located is displayed.

Associated Files

ls.c

Location

Examples/FileSystem

memdebug

Demonstrates the memory debugging subsystem.

Synopsis

memdebug

Description

This program demonstrates the features of the memory debugging subsystem.

The memory debugging subsystem helps you make sure that your programs are freeing all of their memory, not stomping on innocent memory areas, and generally doing illegal things to the Portfolio memory manager. As it detects errors and illegal operations, it displays information in the debugging terminal.

For more information about the memory debugging subsystem, refer to the documentation for the `CreateMemDebug()` function.

Caveats

This program intentionally does some illegal things. Don't do this in your programs!

Associated Files

memdebug.c

Location

examples/Kernel

metronome

Demonstrates how to use the metronome timer feature.

Synopsis

metronome

Description

Demonstrates the use of the metronome timer. The metronome is a mechanism that sends you a signal at a regular interval until you tell it to stop.

The code creates a metronome timer and waits to receive 10 signals from it. The signals are paced at one per second. The pacing is specified when `StartMetronome()` is called.

Associated Files

metronome.c

Location

examples/Kernel

msgpassing

Demonstrates sending and receiving messages between two threads.

Synopsis

msgpassing

Description

Demonstrates how to send messages between threads or tasks.

The `main()` routine of the program creates a message port where it can receive messages. It then spawns a thread. This thread creates its own message port, and a message. The thread then sends the message to the parent's message port. Once the parent receives the message, it replies it to the thread.

Associated Files

msgpassing.c

Location

examples/Kernel

numinfo

Example program used to demonstrate 3DODebug functionality.

Synopsis

numinfo

Description

This program is used to demonstrate functionality that's commonly used in 3DODebug and is referred to in the M2 Debugger documentation. The program counts from 0 to 19 and determines if the value is perfect, a perfect square, a prime number, and how many times it's divisible.

Associated Files

numinfo.c

Location

examples/Miscellaneous/DebugExample

signals

Demonstrates how to use signals.

Synopsis

signals

Description

Demonstrates the use of threads and signals.

The `main()` routine launches two threads. These threads sit in a loop and count. After a given number of iterations through their loop, they send a signal to the parent task. When the parent task gets a signal, it wakes up and prints the current counters of the threads to show how much they were able to count.

Associated Files

signals.c

Location

examples/Kernel

SpriteExample_1

Illustrates how to display a single sprite.

Synopsis

```
SpriteExample_1 <filename.utf>
```

Description

This program demonstrates creation of a GState, allocation of bitmaps, sprite creation, loading a .utf file into a sprite, setting simple texture attributes for a sprite, positioning a sprite, translating a sprite, and drawing a sprite.

Waits for a change in the ControlPad, and then cleans up and exits when a change is detected.

Associated Files

SpriteExample_1.c

Location

TBD

timerread

Demonstrates how to use the timer device to read the current system time.

Synopsis

timerread

Description

This program demonstrates how to use the timer device to read the current system time.

The program does the following:

- * Opens the timer device
- * Creates an IOReq
- * Initializes an IOInfo structure
- * Calls DoIO() to perform the read operation
- * Prints out the current time
- * Cleans up

Note that Portfolio provides convenience routines to make using the timer device easier. For example, CreateTimerIOReq() and DeleteTimerIOReq(). This example intends to show how in general one can communicate with devices in the Portfolio environment. For more information on the timer convenience routines, see the timer device documentation.

Also note that Portfolio provides an alternate higher performance means of sample the current system time. See the documentation on SampleSystemTimeTV() and SampleSystemTimeTT() for more information.

Associated Files

timerread.c

Location

examples/Kernel

timersleep

Demonstrates how to use the timer device to wait for an amount of time specified on the command-line.

Synopsis

```
timersleep <num seconds> <num microseconds>
```

Description

This program demonstrates how to use the timer device to wait for a certain amount of time.

The program does the following:

- * Parses the command-line arguments
- * Opens the timer device
- * Creates an IOReq
- * Initializes an IOInfo structure
- * Calls `DoIO()` to perform the wait operation
- * Cleans up

Note that Portfolio provides convenience routines to make using the timer device easier. For example, `CreateTimerIOReq()`, `DeleteTimerIOReq()`, and `WaitTime()`. This example intends to show how in general one can communicate with devices in the Portfolio environment. For more information on the timer convenience routines, see the timer device documentation.

Arguments

<num seconds>

Number of seconds to wait.

<num microseconds>

Number of microseconds to wait.

Associated Files

timersleep.c

Location

examples/Kernel

Type

Types a file's content to the output terminal.

Synopsis

```
Type -hex {file}
```

Description

Demonstrates how to read a file and send its contents to the Debugger Terminal window using the byte stream routines.

Arguments

-hex

Display the contents of the file in hexadecimal.

{file}

Names of the files to type. You can specify an arbitrary number of file names, and they will all get typed out.

Associated Files

type.c

Location

Examples/FileSystem

Walker

Recursively displays the contents of a directory, and all nested directories.

Synopsis

```
Walker {directory}
```

Description

This program demonstrates how to scan a directory, and recursively scan any directories it contains.

Arguments

```
{directory}
```

Names of the directories to list. You can specify an arbitrary number of directory names, and they will all get listed. If no name is specified, then the current directory in which the ls program is located will get displayed.

Associated Files

```
walker.c
```

Location

```
Examples/FileSystem
```

auto_beat

Automatic rhythm demo that uses lots of AIFF library samples.

Format

auto_beat [pimap file]

Description

Play LOTS of notes pseudo-randomly. Demonstrate direct use of ScoreContext.

Arguments

[pimap file]

Name of a PIMAP file to use. Defaults to auto_beat.pimap.

Associated Files

auto_beat.c, auto_beat.pimap, \$samples

Location

Examples/Audio

capture_audio

Record the output from the DSPP to a host file.

Format

capture_audio [num frames] [dest file]

Description

Captures the output from the DSP into a file on the development station's filesystem. It can be used to check the sound output of a program at important points.

Once the output from the DSP has been captured, you can load it into SoundHack and edit it as follows:

1. From the File menu in SoundHack, choose Open Any.
2. Select the file that was captured with capture_audio.
3. Use the Header Change command and set the captured file's header to a rate of 44100, with 2 channels, and a format of 16-bit Linear.
4. Select Save As command and save the sound file as a 16-bit AIFF format file.

You can then load the AIFF file you created and listen to the captured sounds.

Arguments

[num frames]

Number of sample frames to capture. Defaults to 10000.

[dest file]

Name of the file to save the captured frames to. Defaults to capture.raw. The pathname is a 3DO pathname. The data is written to disk using the `WriteRawFile()`.

Caveats

This program adds an instrument at priority 0. Because the execution order of equal priority instruments depends isn't as predictable as those of differing priority, this program may not be able to capture from other instruments at priority 0.

Associated Files

capture_audio.c

Location

Examples/Audio/Misc

MarkovMusic

Constantly-varying environmentally-aware soundtrack.

Format

MarkovMusic

Description

This program generates a soundtrack from a set of four AIFF data files. The data files can be characterized by their "tension": one is fairly static, the next moves along fairly briskly, the third is more driving still, and the fourth is a transition segment.

The program switches between segments based on a global tension variable controlled by the joypad. The "Up" button increases the tension, the "Down" button decreases it. The "Stop" button terminates the program.

The switching takes place on musical bar boundaries, and the program attempts to switch to a corresponding bar in a four-bar group in the target segment. When tension is decreased, the program transitions through the transition segment.

In addition to switching based on tension, the program branches around within a segment, making branching decisions based on Markov transition tables. This can provide variation for a given tension level.

By following marker naming conventions other AIFF files may be used. Markers are divided into two kinds: major and minor. Major markers (those indicating divisions between types of musical material within a file; for example, theme, variationA, rhythm only) are named

branchx

where x is a number from 1 to the number of major markers. They serve to signal a decision to change tensions, if necessary, and to signal a decision to (possibly) introduce a variation. Each major marker corresponds to a row in the Markov transition tables for the sound.

Minor markers are established wherever a decision to change tensions needs to be made: in the example, this is done on every bar so that the latency between changing tensions with the joypad and hearing the change effected is fairly small. Minor markers are named by appending a counter to the corresponding major marker; for example,

branch2:13

indicates marker 13 in major section 2.

After placing markers, you need to change the constant array SoundNumBlocks to reflect how many major markers you've set in each soundfile.

In addition to placing markers in your soundfiles, you need to alter the transition tables in the associated file markov_tables.c. There is one table for each potential transition, with names that reflect the soundfile to transition from and the soundfile to transition to. For example, the table "mt13" holds probabilities for moving from a marker in soundfile 1 (spA.aiff) to soundfile 3 (spC.aiff). For each major marker in the "from" soundfile, there's a corresponding row in the table. For each major marker in the "to" soundfile, there's a column. The numbers in each column of that row represent the probability that the program will branch from that row's marker to that column's marker. The probabilities for a given row sum to 1.0. For example, if there are four columns in a given table, and for a given row each contains the number 0.25, then there is a 1 in 4 chance that the marker

for that row will branch to any of "to" soundfile's markers, effectively a completely random choice. If one column contains the number 1.0, and the others 0.0, then the program will always branch from that row's marker to that column's marker, a completely determined choice. You can alter the numbers in the tables at any time, so that, for example, every time the program selects a branch, the corresponding probability is decreased and the other columns increased to lessen the likelihood that that branch will happen again.

Associated Files

spA.aiff, spB.aiff, spC.aiff, spD.aiff, markov_music.c, markov_music.h, markov_tables.c, markov_tables.h

Location

Examples/Audio/MarkovMusic

minmax_audio

Measures the maximum and minimum output from the DSP.

Format

minmax_audio

Description

Samples the output of the DSP and returns its maximum and minimum output values. You can use minmax_audio to check that your program outputs are reasonable, non-clipping levels of audio.

The program creates instances of probes from the instruments `minimum.dsp` and `maximum.dsp`. The probes are queried with `ReadProbe()` every five seconds, or `5*240` audio ticks (5 seconds by default).

Signal range is -1.0 to 1.0.

This program runs until it is killed.

Caveats

The loudness of audio is often subjective. Check the loudness of your program by first playing a standard audio CD on a development station, and adjusting the volume of your sound system to a reasonable level. The 3DO program should then produce sounds at a similar volume.

Associated Files

minmax_audio.c

Location

Examples/Audio/Misc

playpimap

Listen to the instruments assigned in a pimap.

Format

playpimap [pimap file]

Description

Play each voice in pimap. This is useful for quickly testing a pimap without having to actually play a song. Playback can be stopped by hitting the 'X' button.

Arguments

[pimap file]

Name of a PIMAP file to use. Required.

Associated Files

playpimap.c, \$samples

Location

Examples/Audio

playsample

Plays an AIFF sample in memory using the control pad.

Format

playsample [<sample file> [rate]]

Description

This program shows how to load an AIFF sample file and play it using the control pad. Use the A button to start the sample, the B button to release the sample, and the C button to stop the sample. The X button quits the program.

The playback rate will default to the sample rate at which the sample was recorded. Thus it should sound normal.

Arguments

<sample file>

Name of a compatible AIFF sample file. If not specified, defaults to loading the standard system sample sinewave.aiff.

[rate]

Sample rate in Hertz (e.g., 22050). Defaults to the sample rate stored in the sample file.

Associated Files

playsample.c

Location

Examples/Audio/Misc

sfx_score

Uses libmusic.a score player as a sound effects manager.

Format

sfx_score

Description

Demonstrate use of ScoreContext as a simple sound effects manager. This gives you dynamic voice allocation, and a simple MIDI-like interface.

To run this program:

1) Install the sample library from the recent 3DO tools disk as documented.

If you don't have the sample library, edit this program to use your own samples. Search for .aiff.

2) Run the program, follow printed instructions.

The idea behind this program is that one can use the virtual MIDI routines in the music library as a simple sound effects manager. You can assign samples to channels, and then play those samples using `StartScoreNote()`. This will take advantage of the voice allocation, automatic mixer connections, and other management code already used for playing scores. You do not have to play sequences or load MIDI files to do this.

You can load your samples based on a PIMap text file using the `LoadPIMap()` routine. This is documented in the manual under score playing. You may also load the samples yourself and put them in the PIMap which will give you more control. This example program shows you how to do that. Samples are assigned to Program Numbers at this step.

Assigning program numbers to channels is accomplished by calling `ChangeScoreProgram()`. Once you have assigned a program to a channel, you can play the sound by calling: `StartScoreNote(scon, Channel, Note, Velocity)`;

Channel selects the sound. Note determines the pitch and may also be used to select parts of a multisample, e.g. a drum kit. Velocity controls loudness. The voice allocation occurs during this call.

Implementation

This example was revised to take advantage of a change made for libmusic.a V22 to permit playing multiple voices per MIDI node, which greatly facilitates sound effects playback.

Location

Examples/Audio/SoundEffects

Notes

1) Samples that are attached to an `InsTemplate` are deleted when the `Template` is deleted. This is caused by setting `AF_TAG_AUTO_DELETE_SLAVE` to `TRUE` in `CreateAttachment`. Set it to `FALSE` if you don't want this to happen.

2) You can share the `ScoreContext` with code that plays a score as long as you don't overlap the channels or program numbers. You might restrict scores to channels 0-7 and programs 0-19. You could then use channels 8-15 and programs 20-39 for sound effects. The advantage of this is that they share the same voice allocation and thus resources will be shared between them. If desired, you could instead

use separate independent ScoreContexts.

See Also

CreateScoreContext()

simple_envelope

Simple audio envelope example.

Format

simple_envelope <Percent Amplitude>

Description

Simple demonstration of an envelope used to ramp amplitude of a triangle waveform. A 3-segment envelope is used to provide a ramp up to a specified amplitude at the start of the sound and a ramp down when the sound is to be stopped. Using envelopes is one technique to avoid audio pops at the start and end of sounds.

Arguments

<Percent Amplitude>

Percentage of full amplitude to play at. Defaults to 100.

Associated Files

simple_envelope.c

Location

Examples/Audio/Misc

ta_attach

Experiments with sample attachments.

Format

ta_attach

Description

Creates a pair of software-generated samples and attaches them to a sample player instrument. The attachments are then linked to one another using `LinkAttachments()` such that they play in a loop.

This demonstrates how multiple attachments can be linked together to form a queue of sample buffers to play and how a client can receive notification of when each sample buffer has finished playing. This is the technique that the Sound Spooler uses to queue sound buffers for playback.

Associated Files

ta_attach.c

Location

Examples/Audio/Misc

See Alsota_spool, `ssplCreateSoundSpooler()`, `LinkAttachments()`

ta_customdelay

Demonstrates a delay line attachment.

Format

ta_customdelay [<sample file> [delay ticks]]

Description

Demonstrates how to create and use a delay line to get real-time echo effects in your program. Loads the specified AIFF file and plays it into a delay line. By tweaking the knobs on the output mixer, you can control the mix of delay sound versus original sound, and the speed at which the echo will die down.

Please note that this example was written before we had the Patch Compiler. If you wish to use reverb or delay effects, I would encourage you to consider using the examples in Examples/Audio/Patches/Reverb.

Arguments

<sample file>

Name of an AIFF file to play.

<delay ticks>

Amount of time, in audio clock ticks, to hold each note. Defaults to 240 ticks.

Associated Files

ta_customdelay.c

Location

Examples/Audio/Misc

ta_envelope

Tests various envelope options by passing test index.

Format

ta_envelope [test code]

Description

Demonstrates creating, attaching, and modifying two envelopes which are attached to triangle instruments.

Arguments

[test code]

Integer from 1 to 13, indicating the number of the test. See the source code for what each test actually does. Defaults to 1.

Controls

A

Starts and releases voice A.

B

Starts and releases voice B.

C

Toggles the states of voices A and B.

Up/Down

Change the time scaling of the envelope.

Associated Files

ta_envelope.c

Location

Examples/Audio/Misc

ta_pitchnotes

Plays a sample at different MIDI pitches.

Format

ta_pitchnotes [<sample file> [duration]]

Description

This program loads and plays the AIFF sample file at several different pitches. It does this by selecting a MIDI note number, which the audio folio maps to a frequency.

Arguments

<sample file>

Name of a sample to be played. The sample should be compatible with `sampler_16_v1.dsp` (16-bit monophonic).

[duration]

Duration of each note, in audio ticks. Defaults to 240.

Associated Files

ta_pitchnotes.c

Location

Examples/Audio/Misc

ta_spool

Demonstrates the libmusic.a sound spooler.

Format

ta_spool

Description

Uses the libmusic.a sound spooler to fill buffers, parse information in the buffers, signal our task when a buffer has been exhausted, and refill the buffers. Use this sample code as a basis for developing your own routines for playing large sampled files, or handling other kinds of buffered data.

Controls

A

Simulates a hold on data delivery. The application continues to consume information in its buffers.

B

Stops the playback function. Releasing it restarts playback. Holding down either shift button causes the spooler to be aborted instead of merely stopped.

C

Enters one-shot mode. Each C button press starts a `ssplPlayData()` sequence. Press the play button returns to normal mode.

Start

Pauses the playback function. Releasing it lets the application continue.

Stop (X)

Quit.

Associated Files

ta_spool.c

Location

Examples/Audio

See Also`ssplCreateSoundSpooler()`, `tsp_spoolsoundfile`

ta_sweeps

Demonstrates adjusting knobs.

Format

ta_sweeps

Description

This program quickly modulates the amplitude and frequency of a sawtooth instrument via tweaking the control knobs repeatedly. The program runs for only a few seconds, and cannot be aborted via the control pad.

Sweep the frequency smoothly as fast as possible to test Knob speed.

Caveats

This specific sound effect could have been performed entirely within the DSP, freeing up the main processor for other purposes.

Associated Files

ta_sweeps.c

Location

Examples/Audio/Misc

See Also

SetKnob(), triangle.dsp

ta_timer

Demonstrates use of the audio timer.

Format

ta_timer

Description

This program shows how to examine and change the rate of the audio clock. It demonstrates use of cues to signal your task at a specific time. It also demonstrates how the audio folio deals with bad audio rate values.

Controls

A

From the main menu runs the audio clock at successively faster rates, using `SleepUntilAudioTime()`.

B

From the main menu uses `SignalAtTime()` in conjunction with cues to wait. While using this method, you can press A or B to abort one of two cues prematurely.

C

From the main menu feeds illegal values to `SetAudioClockRate()`.

Associated Files

ta_timer.c

Location

Examples/Audio/Misc

ta_tuning

Demonstrates custom tuning a DSP instrument.

Format

ta_tuning

Description

Demonstrates how to create a tuning table, how to create a tuning, and how to apply a tuning to an instrument.

Associated Files

ta_tuning.c

Location

Examples/Audio/Misc

tj_canon

Uses the juggler to create and play a semi-random canon.

Format

tj_canon <PIMap> <numVoices>

Description

This program shows how to create and play a simple canon using the juggler and score-playing routines.

It creates a single melodic line, and stores it as a sequence to be repeated on numVoices voices.

Voices can be muted using the A,B,C and direction buttons. The playing can be paused by hitting the 'P' button.

Arguments

<PIMap>

Name of a PIMap to use. This PIMap should have an instrument defined for program number 1 with a "-m x" to set the max number of voices to $x=2*\text{numVoices}$. Eg. "1 /remote/System.m2/Audio/aiff/sinewave.aiff -m 8" Default is "canon.pimap".

Associated Files

tj_canon.c

Location

Examples/Audio/Juggler

tj_multi

Uses the juggler to play a collection.

Format

tj_multi

Description

This program tests the juggler using non-musical events and software-based "timing." It constructs two synthetic sequences based on the TestData data structure, and creates a collection based on these two sequences. It then "plays" the collection via a simple print function, processed at the time of each event in each sequence.

Associated Files

tj_multi.c

Location

examples/Audio/Juggler

tj_simple

Uses the juggler to play two sequences.

Format

tj_simple

Description

This program tests the juggler using non-musical events and software-based "timing." It constructs two synthetic sequences based on the TestData data structure. It then "plays" the sequences via a simple print function, processed at the time of each event in each sequence.

Associated Files

tj_simple.c

Location

examples/Audio/Juggler

tone

Simple audio demonstration.

Format

tone

Description

Plays synthetic waveform for 2 seconds. This demonstrates loading, connecting and playing instruments. It also demonstrates use of the audio timer for time delays.

Associated Files

tone.c

Location

Examples/Audio/Misc

tsp_algorithmic

Advanced sound player example showing algorithmic sequencing of sound playback.

Format

tsp_algorithmic

Description

Assume you have two short sounds in memory and one long sound file with the markers described in Table 1, algorithmically play the sequence described in Table 2.

The technique used to implement this sequence involves the use of static branches, where one segment always leads into another, (e.g. after playing the 1st ending always go back to the loop segment), and decision functions where a conditional branching is required (e.g. from the end of the loop segment either go to the 1st or 2nd ending).

This also demonstrates using sounds spooled from disc and played directly from memory.

Tables

Table 1. Long Sound File Markers

Marker	Description
-----	-----
BEGIN	beginning segment ("attack")
Loop	loop segment ("sustain")
First Ending	1st ending segment ("gap")
Second Ending	2nd ending segment ("release, end")
END	

Table 2. Sound Sequence

Description	What you should hear
-----	-----
long sound beginning segment	"attack"
long sound loop segment	"sustain"
long sound 1st ending segment	"gap"
long sound loop segment	"sustain"
long sound 2nd ending segment	"release, end"
short sound 1	<honk>
long sound loop segment	"sustain"
long sound 1st ending segment	"gap"
long sound loop segment	"sustain"
long sound 2nd ending segment	"release, end"
short sound 2	<blap>

Associated Files

tsp_algorithmic.c, words.aiff

Location

Examples/Audio/SoundPlayer

See Also

```
spCreatePlayer(),tsp_switcher,tsp_rooms
```

tsp_rooms

Room-sensitive soundtrack example using advanced sound player.

Format

tsp_rooms

Description

Creates a thread to playback a sound track based on a global room variable gRoom.

The parent task, tsp_rooms, gets control port events and updates gRoom as described below.

The soundtrack thread, Soundtrack, uses the advanced_sound player to play a unique sound file for each room. When the main task changes rooms, the soundtrack thread adapts the soundtrack to the change in room at a musically convenient location.

The way this is done is that each room's soundfile is designed so it can play in a loop. In addition to this, the end of each sound file, and additional optional marked locations within the sound file, must be musically sensible locations to transition to the beginning of any of the other room's sound files. When playback reaches the next marked position in, or the end of, the current room's sound, a default decision function is called to check to see if the main task changed room. If the room is still the same, then the current sound continues to play, or loops if at the end.

If the room has changed, the SPPlayer is instructed to begin playing the sound for the new room.

Note that the soundtrack adapts to room changes by the main task instead of making a sharp transition as soon as the room changes. It is possible for the main task to make several room changes before the soundtrack thread discovers that the room has changed at all. This results in smooth soundtrack transitions that occur at seemingly random locations, which is far less likely to become annoying to a game player than a soundtrack that predictably changes the instant the player crosses a given threshold.

Controls

A

Enter room 0.

B

Enter room 1.

C

Enter room 2.

X (Stop)

Exit when the sound for the current room gets to a marker.

Shift-X

Exit immediately.

Associated Files

tsp_rooms.c, words.aiff

Location

Examples/Audio/SoundPlayer

See Also

`spCreatePlayer()`, `tsp_switcher`, `tsp_algorithmic`

tsp_spoolsoundfile

Plays an AIFF sound file from a thread using the advanced sound player.

Format

`tsp_spoolsoundfile <sound file> [num repeats]`

Description

Plays an AIFF sound file using a thread to manage playback. The file is played as a continuous loop.

Arguments

`<sound file>`

Name of an AIFF file to play.

`[num repeats]`

Optional number of times to repeat the sound file. Defaults to 1.

Associated Files

`tsp_spoolsoundfile.c`

Location

`Examples/Audio/SoundPlayer`

See Also

`spCreatePlayer()`, `tsp_algorithmic`, `tsp_rooms`

tsp_switcher

Advanced sound player example that switches between sounds based on control pad input.

Format

tsp_switcher

Description

Loops one of three sound files off of disc. The user can select a different sound to loop by pressing the A, B, or C buttons on the control pad. The last button pressed corresponds to the sound being played.

This program demonstrates how one might use the advanced sound player as an engine for doing environmentally-sensitive soundtrack playback. Note that the sound being played doesn't change to the newly selected one until the end of the current sound is reached. This demonstrates, for example how a score might be made to change at the next musically sensible point. See `tsp_rooms` for another way to do this.

Controls

- A
Select sound #1.
- B
Select sound #2.
- C
Select sound #3.
- X (Stop)
Quit when done playing.
- Shift-X
Quit immediately.
- Start
Toggle pause on/off.

Associated Files

tsp_switcher.c

Location

Examples/Audio/SoundPlayer

See Also

`spCreatePlayer()`, `tsp_rooms`, `tsp_algorithmic`

windpatch

Creates a "wind" sound effect using the audio folio's patch compiler.

Format

windpatch

Description

Creates and plays a howling wind Patch Template. This demonstrates use of the audio folio's Patch Compiler.

This program runs the X button is pressed.

Associated Files

windpatch.c

Location

Examples/Audio/SoundEffects

See Also

PatchCmd

tb_envelope

Trigger envelopes using the Beep folio.

Format

tb_envelope

Description

Play a sample and control its envelope using the Beep Folio. Up and Down buttons cause program to switch to higher or lower DMA channels.

Button A triggers a fast attack and a slow release. Button B triggers a slow attack and a fast release. Button C forces the amplitude to one then starts a slow envelope to zero. May pop.

Associated Files

<:beep:beep.h> <:beep:basic_machine.h>

Location

Examples/Audio/Beep

tb_playsamp

Play a sample using the Beep Folio.

Format`tb_playsamp {samplename}`**Description**

Play a sample using the Beep Folio. Up and Down buttons cause program to switch to higher or lower DMA channels.

A,B and C buttons can be played like a Trumpet. RightShift button shifts up an octave.

Tuning is based on the following 7 ratios: 1/1 9/8 5/4 4/3 3/2 5/3 7/4

Associated Files

`<:beep:beep.h>` `<:beep:basic_machine.h>`

Location

Examples/Audio/Beep

tb_spool

Spool audio from memory using the Beep folio.

Format

tb_spool

Description

Spool continuous audio to the beep folio. Plays random tones until stopped. Up and Down buttons cause program to switch to higher or lower DMA channels. Demonstrate use of `SetBeepChannelDataNext()` with signal. This technique can be used to spool audio off of disk.

Associated Files

<:beep:beep.h> <:beep:basic_machine.h>

Location

Examples/Audio/Beep

cpdump

Queries the event broker and prints out a summary of what's connected to the control port.

Format

cpdump

Description

Sends three messages to the event broker (EB_DescribePods, EB_GetFocus, and EB_GetListeners) and prints out the values that the event broker returns for these messages.

Associated Files

cpdump.c

Location

Examples/EventBroker

focus

Talks to the event broker and switches the focus to a different listener.

Format

focus [listener]

Description

Lets you view the current focus holder, or change it.

Arguments

[listener]

Name or hexadecimal address of a message port to which the focus should be diverted. If this is not specified, the program lists the current focus holder.

Associated Files

focus.c

Location

Examples/EventBroker

lookie

Connects to the event broker and reports any events that occur.

Format

lookie [focus | hybrid]

Description

Connects to the event broker and requests to be informed about every event. Whenever any event occurs, the important data from the event is displayed in the Debugger Terminal window.

Arguments

[focus]

Makes lookie a focus listener.

[hybrid]

Makes lookie a hybrid listener.

Associated Files

lookie.c

Location

Examples/EventBroker

luckie

Uses the event broker to read events from the first control pad.

Format

luckie [anything]

Description

Uses the event broker to monitor and report activity for the first control pad plugged in to the control port.

Arguments

[anything]

If you supply no arguments to this program, it asks `GetControlPad()` to put the task to sleep when waiting for an event. If you supply an argument, `GetControlPad()` will not put the task to sleep, and the program will poll the control pad.

Associated Files

luckie.c

Location

Examples/EventBroker

maus

Uses the event broker to read events from the first mouse.

Format

maus [anything]

Description

Uses the event broker to monitor and report activity for the first mouse plugged in to the control port.

Arguments

[anything]

If you supply no arguments to this program, it asks `GetMouse()` to put the task to sleep when waiting for an event. If you supply an argument, `GetMouse()` will not put the task to sleep, and the program will poll the mouse.

Associated Files

maus.c

Location

Examples/EventBroker

DrawLines

Illustrates three methods of drawing lines.

Synopsis

DrawLines

Description

This program demonstrates how to draw single and multiple lines and how to call `F2_DrawLine()`, `F2_ColoredLines()`, and `F2_ShadedLines()`.

Associated Files

DrawLines.c

Location

Examples/Graphics/Frame2d

MoveSprite

Illustrates how to display, move, rotate and scale a single sprite.

Synopsis

```
MoveSprite <filename.utf>
```

Description

This program loads a sprite and lets you move, scale, and rotate it. Also demonstrates how to do GState double-buffering.

The joystick moves the sprite. A enlarges the sprite, B shrinks it. Left and right shift rotate the sprite. C restores the sprite to its original size and location. Stop quits the program.

Associated Files

MoveSprite.c

Location

Examples/Graphics/Frame2d

Points

Illustrates how to draw points.

Synopsis

Points

Description

Demonstrates how to draw points with the 2D API.

Associated Files

Points.c

Location

Examples/Graphics/Frame2d

Rectangles

Illustrates how to move and map the corners of a sprite.

Synopsis

Rectangles

Description

Shows two methods of drawing rectangles with the 2D API. The first simple sets coordinates and a color. The second draws whatever is found at a given area in memory.

Associated Files

Rectangles.c

Location

Examples/Graphics/Frame2D

RenderOrder

Illustrates how to link sprites and control their rendering order.

Synopsis

```
RenderOrder <filename.utf> [ <filename.utf> <filename.utf> ... ]
```

Description

This program loads up to ten textures, puts them into a list, and draws them all with a call to F2_DrawList.

The A button cycles control of the sprites. The directional pad moves the sprite. The B and C buttons move the sprite forward and backward through the list.

Associated Files

RenderOrder.c

Location

Examples/Graphics/Frame2d

SimpleSprite

Illustrates how to display a single sprite.

Synopsis

```
SimpleSprite <filename.utf>
```

Description

This program demonstrates creation of a GState, allocation of bitmaps, sprite creation, loading a .utf file into a sprite, setting simple texture attributes for a sprite, positioning a sprite, translating a sprite, and drawing a sprite.

Waits for a change in the ControlPad, and then cleans up and exits when a change is detected.

Associated Files

SimpleSprite.c

Location

Examples/Graphics/Frame2D

spin3d2d

Rotate 2d and 3d objects together

Synopsis

```
spin3d2d [InitGP params] file.bsf file.utf
```

Description

Given a binary SDF file and a UTF file, load the SDF object and spin it in 3 space, while moving and spinning the UTF texture as a sprite. Also, move the sprite's Z coordinate in and out so that it moves through the SDF object. Control pad actions will exit the program.

Arguments

[InitGP params]

-?: Display help. -b: Enable destination blending. -640: Wide display. -480: Tall display. -32: True color (32 bits per pixel). -h: Hi-res (640x480x32bpp) shortcut. -s#: Use # frame buffers (default = -s2).

file.bsf

Name of binary SDF file to load. The last object defined within the world class structure will be used.

file.utf

Name of the texture to load into the 2D sprite.

Associated Files

spin3d2d.c

Location

Examples/Graphics/Frame2d

scenepf

Measure the performance of scenes.

Format`scenepf [model ...]`**Arguments**

All arguments are compiled SDF files. They may contain animation engines, which are used to animate the scene. The name of the models and engines must be related to the name of the file. (See below for the relationship.)

Description

The file name is stripped of the extension to get the "base name". For example, `skeleton.bsf` would have a base name of `skeleton`. The group (model) in the file is expected to be `basename_World` (e.g. `skeleton_World`). If an animation is present in the file, it is expected to be named `basename_KfEngines` (e.g. `skeleton_KfEngines`). This is consistent with the naming conventions used in the current conversion tools.

The models are gathered into two umbrella groups: those without animations (the static group) and those with animations (the dynamic group). The static group is placed in the scene. The dynamic group is also placed in the scene, and is animated.

While running, `scenepf` will print out the frame rate on the debugger window periodically. To minimize the time spent printing, the program should be run with special mode turned on (`command=-`).

`Scenepf` will exit when the stop key is pressed.

The program focuses the control pad in one of three ways: camera, pivot, and walkers (the dynamic group). The start key is used to switch between focuses. The new focus is printed on the debugger output window. The controls work as follows:

Camera Focus

The direction keys move the camera in the indicated direction (along the camera's X-Y plane). When the A key is pressed, movement is in the camera's X-Z plane.

When the B key is pressed, the direction keys rotate the camera about its X and Y axis.

The right and left shift keys rotate the camera about its Z axis.

When the C key is pressed, the direction keys rotate the camera about the pivot point. The pivot stays in the same relative position to the camera.

Pivot Focus

When in pivot mode, a small, colored box appears on the screen to indicate the pivot point.

The controls for the pivot work in the same manner as for the camera, except the C key works the same as the B key. In addition, it should be noted that rotating the pivot has no effect except for the appearance of the pivot.

Walker Focus

The A key will cause a new group of walkers to be added into the scene, immediately behind the last group. There is a limit to the number of walker groups that can be added into the scene.

The C key will remove the last group of walkers from the scene. The original set of walkers cannot be deleted.

As walkers are added and deleted from the scene, a message will be printed indicating the number of walkers present.

Location

Examples/Graphics/Frame/sceneparf

m2perf

Measure the raw performance of the M2

Format`m2perf [option ...]`**Description**

Render a large number of triangles, measuring the time that it takes to render. The results are reported on stdout. The program first outputs a header summarizing all the total options, both those specified and those derived from defaults. A reporting header is output. Then tests are run, and the results of each test are reported, one line per test.

The options (and their arguments) are not case sensitive. The order in which options and arguments are specified should not affect the outcome of the tests.

Many options take an argument. The argument is either a set of predefined strings, abbreviations of the strings, or numeric values. The description of the option will specify the valid arguments. If a numeric option is set to 0, the default value will be used.

Most options may be specified more than once. Options which require a numeric argument will indicate if they can be specified more than once. When an option is specified more than once, a test is run with each argument, while all other options are held constant. Exceptions are noted in the options descriptions.

As a shorthand notation, multiple arguments may be specified by separating them with a comma. Thus, the option:

`-cull back,front`

is equivalent to:

`-cull back -cull front`

If an options may be specified more than once, the special argument All may be used to specify that all variants that make sense should be run. For options that require a numeric argument, the argument All indicates that a predetermined set of number will be used. The description of the option will indicate the values.

Many options that do not take an argument specify a boolean flag. The reverse of the option has the prefix 'no'. Both options may be specified, resulting on the benchmark being run for each.

When multiple options are specified, each with multiple arguments (or both variants for boolean options), the benchmark will be performed for each possible combination. For example:

`m2perf -zbuf -nozbuf -cull all`

will run a total of six (6) tests combining every variant of -zbuf/-nozbuf and -cull.

The default for each options is specified with that option. The default options are only used when an option is not specified; they do not add to the options list.

Arguments

Reporting Control

-outfile <filename> [Not implemented]
Generate the report in the specified file.

The default name is m2perf.out.

-dumpfile <filename> [Not implemented]
Dump all packets sent to the hardware in the specified file. The file can then be analyzed by external tools, or played back for hardware performance measurement.

If not specified, no dump file is created.

Rendering Options

-buffering <buffertype>
Specifies the type of hardware buffering to use. <buffertype> may be one of:

Single
Double
Triple [Not implemented]
Null

The Null buffer type indicates the the command buffer should not be sent to the hardware. When -dumpfile is specified, the dump file is still created.

The default buffer type is Double.

-clip <clipOpt>
Specify how the scene is to be clipped. <clipOpt> may be one of:

None
Visible
Partial
Hidden

None indicates that no clipping is to be performed. Visible will place the scene entirely on-screen, and enables the clipping code. Partial rotates the camera so that about half the scene is off-screen. Hidden rotates the camera so that the scene is completely off-screen.

The default clip mode is None.

-cull <cullSide>
Specifies which culling option to use. <cullSide> may be one of:

Back
Front
None

The default cull option is Back.

-facetSize <size>
Specifies how large the triangles rendered should be. <size> is a symbolic name for the average number of pixels that <size> should be one of the following (pixel size is indicated in parenthesis):

Small (10)
Medium (20)
Large (40)

The default size is Small.

-material <matType>

Specify which material type properties are used with lit geometry (-surf includes Normal). The material type properties defined are:

Diffuse
Specular
Emissive
TwoSided

<matType> may be abbreviated to D (diffuse), S (specular), E (Emissive), and T (TwoSided). Multiple properties may be specified for the same scene by using the abbreviations for all desired properties, e.g, DST for Diffuse, Specular, and TwoSided. The letters may appear in any order.

The argument All is equivalent to specifying all 15 combinations of surfaces.

Note: DST is different from D,S,T. The former specifies one test, with all three properties enabled simultaneously. The latter specifies three tests, each with only one of the properties enabled.

The default material type property is Diffuse.

-perspective

-noperspective

Specifies whether to perspective correct textures (with GP_PerspCorrect).

The default is -perspective.

-surf <surfType>

Specifies the surface type to be used for color calculations. <surfType> should be one of:

Color
Normal
Texture
ColorPerFacet
NormalPerFacet
ColorTexture
NormalTexture
ColorTexturePerFacet
NormalTexturePerFacet

<surfType> may be abbreviated using the letters C (color), N (normal), T (Texture), and P (PerFacet). Combinations are specified by listing the letters corresponding to all the surface types desired, e.g, NTP for ColorTexturePerFacet. The letters may appear in any order.

The argument all is equivalent to specifying all the combinations listed above.

Note: NT is different than N,T. The former species one test run, with a surface type of ColorTexture. The latter specifies two test runs, with surface types of Color and Texture, respectively.

Note: PerFacet is a modifier for Color and Normal only. It can only be specified when either Color or Normal is specified. It indicates that only one color or normal is to be used for each triangle (flat shading). PerFacet does not affect Texture.

Note: Color and Normal are mutually exclusive.

The default surface type is NormalTexture.

-zbuf

-nozbuf

Specifies whether Z buffering should be enabled.

The default is -zbuf.

-quadmesh

-trifan

-trilist

-trimesh

-tristrip

Specifies which primitive to run.

Quadmesh: A rectangular surface with a single quadmesh is constructed. It measures $(\text{triPerPrimitive} + 1) / 2$ by $\text{primitivesPerSurface}$. (One extra set of coordinates is added in each dimension.) Texture coordinates are generated that spread a texture over the complete surface. The entire surface is colored uniformly for pre-lit cases.

Trifan: [Not implemented]

Trilist & Tristrip: A horizontal list of triangles triPerPrimitive long is generated. It is attached to a surface $\text{primitivesPerSurface}$ times, with a translation along the Y axis occurring each time. Texture coordinates are generated for the original strip, and are adjusted in the geometry before it is added to the surface. Color values for the pre-lit case are generated for the original strip, and are not adjusted.

Trimesh: The data for the quadmesh is generated. Strips are built to emulate the tristrip test data. The resulting surface looks like the quadmesh when triPerPrimitive is even.

After the surface is generated, it is attached to a model. The model is then cloned, translated a small amount along the Z axis (alternating between +Z and -Z), and attached to the hierarchy modelsPerScene times.

The default is -trimesh.

-allprims

Equivalent to specifying -quadmesh -trifan -trilist -trimesh -tristrip.

-animate [Not implemented]

Specifies that after running a test, the test is to be re-run, and the scene is to be moved between frames. This measures the effect of changing the transforms.

Animation is not enabled by default.

-bufsize <bufSize>

Specifies a buffer size, in kilobytes, to use when communicating with the M2. If multiple sizes are specified, all tests will be repeated with each buffer size. There is a limit on the number of buffer sizes that may be specified.

The argument All is equivalent to specifying '24,48,128'.

The default buffer size is 128k.

-light <lightType>

Specifies a light that should be placed in the scene. <lightType> should be a combination of:

None
Directional
Point
Spot
SoftSpot

Specifying more than one type is given for a light (e.g. -light Directional,Spot), indicates that a single light should vary between the specified types. Multiple -light options indicate multiple lights in the scene. The total number of runs is increased so that the lights cycle through all the combinations specified. In other words:

-light Point,Spot -light Directional,Spot

will run four (4) unique tests, each with two lights, where the first light is either Point or Spot, and the second light is either Directional or Spot.

The lights are added into the scene before any geometries.

Note: The None setting may be used to disable a light for a test run.

Note: The order of the lights shouldn't affect the outcome.

The default light setting is a single Directional light (i.e. -light Directional)

-triPerPrimitive <count>

-primitivesPerSurface <count>

-modelsPerScene <count>

Specifies the number of triangles in a primitive, the number of primitives per surface, and the number of models in the scene. See the drawing primitive descriptions to see how these are used.

-texloadsPerScene <count>

Specifies the number of texture loads that will occur in the scene. The texture loads will be evenly spaced among the models. SURF_UseLast will be used for the texture index for intermediate geometry and models. This option does not affect the scene when texturing is not enabled.

The default number of texture loads is 5.

-numMaterials <count>

Specifies the number of different materials to use when lighting is enabled. <count> different materials are created, and primitives are assigned consecutive material indices. If the number of materials is less than the number of primitives in the surface, the indices cycle back to the beginning.

Note: This option may only be specified once.

The default material count is 1.

-hierarchyDepth <count> [Not implemented]

Specifies the depth of the hierarchy of models. The models in the scene will be distributed among the levels of the hierarchy. Increasing this number increases the amount of hierarchy that the code must traverse before getting to the geometry.

Note: This options may only be specified once.

The default hierarchy count is 1.

-all

Run as though every boolean option was specified with both variants, and every non-boolean option was specified with All. This is equivalent to specifying:

```
m2perf
-buffering all
-clip -noclip
-cull all
-facetSize all
-material all
-perspective -noperspective
-surf all
-zbuf -nozbuf
-allprims
-animate
-bufsize 24,48,128
-light all -light all    (Note: Two lights)
```

Location

Examples/Graphics/Frame/m2perf

DataPlayer

A DataStream DATA subscriber example program.

Format

```
DataPlayer [-l[oop]] <streamFile>
```

```
Control Pad functions:
```

```
  "A"      button: rewinds the stream  
  "B"      button: and up/down arrows cycle through the channel  
displayed  
  "C"      button: toggles on-screen data display  
  "Stop"   button: exits the program
```

Description

This is an example M2 application that uses DataStreaming components to playback a stream containing DATA subscriber data.

Location

```
{3doreMOTE}/Examples/Streaming/DataPlayer
```

EZFlixPlayer

A `DataStream` example program that plays synchronized video and audio.

Format

```
EZFlixPlayer [-16 | -32] [<stream filename>]
-16 for 16 bits/pixel. -32 for 32 bits/pixel (the default).
```

Control Pad functions:

<code>[]</code>	Stop
<code>>/ </code>	Pause/Resume
<code>LeftArrow</code>	Jump to start of stream
<code>UpArrow</code>	"Cut" to start of stream without flushing
<code>LeftShift</code>	Toggle H pixel averaging
<code>RightShift</code>	Toggle V pixel averaging

Description

This is an example 3DO application that uses `DataStreaming` components to playback a stream containing EZFlix video synchronized with `SquashSound` audio data.

Location

`{3doremote}/Examples/Streaming/EZFlixPlayer`

PlaySA

A DataStream example program that plays streamed audio.

Format

PlaySA <streamFile>

Where <streamFile> is a woven DataStream file containing SquashSnd audio data.

Control Pad functions:

>/|| button: Pause/resume

Stop button: Stop and exit

C button: Rewind the stream

B button: Cycle the channel number (0-3) for volume & pan adj

UpArrow: Increase the volume of the current channel

DownArrow: Decrease the volume of the current channel

LeftArrow: Pan the current channel left

RightArrow: Pan the current channel right

LeftShift+A: Mute/unmute channel 0

LeftShift+B: Mute/unmute channel 1

LeftShift+C: Mute/unmute channel 2

RightShift+C: Mute/unmute channel 3

Description

This is an example 3DO application that uses DataStreaming components to playback a stream containing SquashSnd audio data. If the streamed audio file does not contain a streamed header, the default stream header is used. The defaults are:

```

streamblocksize 32768 (size of stream buffers)
streambuffers      4 (suggested number of stream buffers to
use)
streamerDeltaPri  -10 (delta priority of streamer thread)
dataacqdeltaPri   -9 (delta priority for data acquisition
thread)
audioclockchan    0 (logical channel number of audio clock
channel)
enableaudiomask   1 (enables logical audio channel 0)

subscriber SNDS    10 (delta priority of audio subscriber
thread)
preloadinstrument  SA_22K_16B_M_SQD2,
                   SA_44K_16B_M
                   (preload instrument tags:
                   22KHz, 16 bit mono, 2 to 1 compresses,
                   44KHz, 16 bit mono)
```

Location

{3doremote}/Examples/Streaming/PlaySA

Caveats

You will get "Error sending control message" error message if you hit the mute button on a channel that is not the current channel.

VideoPlayer

A DataStream example program that plays synchronized video and audio.

Format

```
VideoPlayer [-16 | -24] [-1 | -loop] [-?] [<streamFile>]
  -16 for 16 bits/pixel. -24 for 24 bits/pixel. Default is 24.
  -1 or -loop to loop until the STOP key is pressed.
  -? to just print usage info.
  The default <streamFile> is "video1.stream".
```

Control Pad functions:

[]	Stop
>/	Pause/Resume
Up	Jump forward to the next marker (requires markers in the stream)
Down	Jump backward to the previous marker (requires markers)
Left	Jump to start of stream ASAP
Shifted-Left	'Cut' to start of stream

Description

This is an example M2 application that uses DataStreaming components to playback a stream containing synchronized video and audio data.

Location

{3doreMOTE}/Examples/Streaming/VideoPlayer

Chapter 2

Shell Commands

This section presents the reference documentation for the shell commands used during development.

AcroAdmin

Acrobat Filesystem Administration Utility

Format`AcroAdmin [] dev_name offset mount_dir`**Description**

XXX - DO THE DOCS

Arguments

XXX - DO THE DOCS

`dev_name`

Name of the device the filesystem resides on.

`offset`

Offset within the device where the filesystem resides.

`mount_dir`

Name of the directory the filesystem is mounted on.

Implementation

Command implemented in V27.

Location

System.m2/Programs/AcroAdmin

See Also

AcroFormat

AcroFormat

Format acrobat filesystems.

Format

AcroFormat Options dev_name offset mount_dir

AcroFormat [-b blockSize]
 [-d dirSize]
 [-e extentSize]
 [-I iperiblk]
 [-i numInodes]
 [-n]
 [-M]
 [-m]
 [-S inodeSize]
 [-s fsSize]
 [-u uniqueID]
 <devName> <offset> <mountName>

Clock

Gets/sets the date and time from the battery-backed clock.

Format

clock [-date <dd-mmm-yyyy>]
[-time <hh:mm:ss>]

Description

This command lets you set or get the time and date of the system's battery-backed clock.

If you run the command with no argument, it just prints out the current clock settings. Specifying either the -date or -time argument will actually change the values in the clock accordingly and will display the new time.

Arguments

-date <dd-mmm-yyyy>

Lets you specify a new date for the clock. The day of the month is specified in numerals, the month is specified as a three letter string, and the year is a four character numeral. For example: "29-Feb-1996"

-time <hh:mm:ss>

Lets you specify a new time for the clock. You specify the hours, minutes, and seconds. For example: "12:34:56".

Location

System.m2/Programs/Clock

Copy

Copies files or directories

Format

Copy <file1> <file2> Copy <directory1> <directory2> Copy {file | directory} <directory>

Description

In the first form, copy copies the content of filename1 to filename2. If filename2 exists, appropriate error is returned. In the second form copy recursively copies all content of directory1 to directory2. The destination directory must already exist.

In the third form the content of each filename or directory in the argument list is copied to the destination directory. The destination directory must already exist.

WARNING: Beware of a recursive copy of a parent directory into its child. For example:

```
Copy /remote/mydir /remote/mydir/mysubdir
```

This would fill up the file system.

Implementation

Command implemented in V20.

Location

System.m2/Programs/Copy

See Also

Delete, Mkdir, Rmdir

Delete

Deletes files and directories.

Format

Delete [-all] {file | directory}

Description

This command lets you delete files or directories. Without the -all option, this command will delete only files and empty directories. When the -all option is given, the command will recursively delete directories and their contents.

Arguments

-all

Specifies that if a non-empty directory is provided, all of its contents may be deleted. If this option is not provided, then attempting to delete a directory that is not empty will fail.

{file | directory}

Specifies the name of the file or directory to delete. Any number of files or directories can be specified.

Implementation

Command implemented in V20.

Location

System.m2/Programs/Delete

Dismount

Dismounts a file system.

Format

Dismount <mountName>

Description

This command lets you dismount file systems, which produces similar results as physically removing the media containing the file system.

Arguments

<mountName name>

The name of the file system to dismount.

Implementation

Command implemented in V20.

Location

System.m2/Programs/Dismount

See Also

Mount

dumplogs

Dump event logs to the debugging terminal.

Format

dumplogs

Description

This command lets you display the contents of the Lumberjack event logs to the debugging terminal. You use the log command to initiate event logging.

Refer to the Kernel documentation for more information on Lumberjack.

Implementation

Command implemented in shell V27.

Location

Built-in shell command.

expunge

Remove unused demand-loaded modules from memory.

Format

expunge

Description

This command causes any demand-loaded modules that have a 0 use count to be removed from memory.

Implementation

Command implemented in shell V27.

Location

Built-in shell command.

FileAttrs

Gets or sets attributes of a file.

Format

```
FileAttrs [-version <8 bit version>]
          [-revision <8 bit revision>]
          [-type <32 bit file type number>]
          {file}
```

Description

This command lets you set attributes of a file. The supported attributes include its version and revision codes, as well as its 4 byte file type value.

If you run the command without specifying any attributes, it will display the current attributes of the files.

Arguments

-version <8 bit version>
Specifies the 8 bit version code for the files.

-revision <8 bit revision>
Specifies the 8 bit revision code for the files.

-type
Specifies the 32 bit file type value to set.

{files}
Specifies the names of the files to get or set the attributes of. Any number of files or directories can be specified.

Implementation

Command implemented in V27.

Location

System.m2/Programs/FileAttrs

FindFile

Searches for a file, and prints its pathname.

Synopsis

```
FindFile <partialPath>
```

Description

Scan for a file and print its pathname.

Arguments

partialPath

Specifies a relative pathname. FindFile will search for the file identified by this path in the current directory, and in the root directory of each mounted filesystem. If more than one such file exists, the file with the highest version/revision number will be chosen.

Location

System.m2/Programs/FindFile

FSInfo

Displays information on mounted file systems.

Format

FSInfo

Description

This command displays information about mounted file system. The information includes the raw device the file system is running on, the FS creation time, the FS size, and its status. The status specified whether the FS is read-write or read-only, whether it is online, and whether it is intended for end-user storage.

Implementation

Command implemented in V27.

Location

System.m2/Programs/FSInfo

HW

Prints a list of all hardware resources currently in the system.

Format

HW

Description

This command prints out a list of all hardware resources currently in the system. Hardware resources are a data structure built on bootup and rebuilt when new devices go offline or come online. These structures define which hardware is currently attached.

Implementation

Command implemented in V29.

Location

System.m2/Programs/HW

Intl

Gets or sets the international settings of the machine.

Format

```
Intl [-code [German |  
          Japanese |  
          Spanish |  
          Italian |  
          Chinese |  
          Korean |  
          French |  
          UKEnglish |  
          AusEnglish |  
          MexSpanish |  
          CanEnglish ]
```

Description

This command lets you display the current country and language as reported by the international folio. The information displayed corresponds to the country codes and language codes listed in the `<:international:intl.h>` include file.

By using the `-code` option, you can adjust the machine's international setting. You supply the name of the international setting you want. The system's language and country codes will be adjusted accordingly.

Arguments

`-code <setting>`
Changes the machine's international setting.

Implementation

Command implemented in V24.

Location

System.m2/Programs/Intl

Items

Displays lists of active items

Format

```
Items [item number]
      [-name <item name>]
      [-type <item type>]
      [-owner <task name>]
      [-full]
```

Description

This command displays information about the items that currently exist in the system. The information is sent out to the debugging terminal.

Arguments

[item number]
Requests that only information on the item number supplied be displayed. The item number can be in decimal, or in hexadecimal starting with 0x or \$.

-name <item name>
Requests that only items with the supplied name be listed.

-type <item type>
Requests that only items with the supplied type be listed. The currently supported types are:

Folio Task FIRQ Semaphore Message MsgPort Driver IOReq Device Timer ErrorText
FileSystem File Alias Locale Template Instrument Knob Sample Cue Envelope
Attachment Tuning Probe AudioClock Bitmap View ViewList TEContext Projector Font

-owner <task name>
Requests that only items owned by the supplied task be displayed.

-full
Requests that any extra information available on the items being listed be displayed.

Implementation

Command implemented in V20.

Location

System.m2/Programs/items

killtask

Remove an executing task or thread from the system.

Format

killtask <task name | task item number>

Description

This command removes a task or thread from the system. When removing a task, all resources used by the task are also returned to the system.

Arguments

<task name | task item num>

This specifies the name or item number of the task to remove. The item number can be in decimal, or in hexadecimal starting with 0x or \$.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

kill

See Also

showtask

log

Control event logging.

Format

log {[-]eventType}

Description

This command lets you control Lumberjack, the Portfolio logging service. Lumberjack is used to log system events to aid during debugging. The kernel uses Lumberjack to store a large amount of information about what is currently happening in the system.

To display the contents of the logs to the screen, use the `dumplogs` command.

Refer to the Kernel documentation for more information on Lumberjack.

Arguments

{[-]eventType}

This argument lets you specify the types of event to be logged. You can specify any number of events at the same time. The possible event types are: user, tasks, interrupts, signals, messages, semaphores, pages, items, ioreqs. If you put a - in front of an event type, it turns off logging for that event. If you do not specify any event type, the command simply displays the events currently being logged.

Implementation

Command implemented in shell V27.

Location

Built-in shell command.

Ls

Displays the contents of a directory.

Synopsis

```
Ls {directory}
```

Description

Scan directories and display their contents.

Arguments

```
{directory}
```

Name of the directories to list. You can specify an arbitrary number of directory names. If no name is specified, the current directory is displayed.

Location

System.m2/Programs/Ls

MinimizeFS

Minimize filesystem/folio memory footprint.

Synopsis

MinimizeFS

Description

MinimizeFS uses the `MinimizeFileSystem()` call to ask the Portfolio filesystem manager to minimize its memory footprint.

Implementation

Command implemented in V29.

Location

System.m2/Programs/MinimizeFS

See Also

Mount, Dismount

MkDir

Creates new directories.

Format

MkDir {directory}

Description

This command creates directories. It can only create directories in the filesystems that are mounted read-write and support directories.

Arguments

{directory}

Names of the directories to create.

Implementation

Command implemented in V29.

Location

System.m2/Programs/MkDir

See Also

Delete

Mount

Mounts a file system.

Format

Mount <device name> [offset]

Description

This command lets you mount file systems that aren't automatically mounted by the system. Mounting them prepares them for use and makes them available for regular file I/O operations.

Arguments

<device name>

This specifies the name of the device the file folio should look on to find a file system.

[offset]

The block offset within the device where the file system label information can be found. If this argument is not specified, offset 0 is assumed.

Implementation

Command implemented in V20.

Location

System.m2/Programs/Mount

See Also

Dismount

MountLevel

Displays or changes the current filesystem automounter level

Synopsis

```
mountlevel [newlevel]
```

Description

Displays the current filesystem mount level, and optionally changes it to a different level.

Arguments

[newlevel]

New mount level (an integer between 0 and 255) --

Location

System.m2/Programs/mountlevel

Options

Controls various system run-time options.

Format

Options [-icache <onloff>]
 [-dcache <onloff>]
 [-printf <onloff>]
 [-prefetch <onloff>]
 [-ldebug <onloff>]

Description

This command lets you adjust system options. When run with no argument, it reports the current state of some of the options it controls.

Arguments

- icache <onloff>
 Turn the CPU instruction cache on or off.
- dcache <onloff>
 Turn the CPU data cache on or off
- printf <onloff>
 Turn debugging output on or off.
- prefetch <onloff>
 Turn CPU prefetch on or off.
- ldebug <onloff>
 Turn launch debugging on or off.
- memdebug <onloff>
 Provides an command line interface to CreateMemDebug

Implementation

Command implemented in V27.

Location

System.m2/Programs/options

ProxyFile

Proxy a file so that it becomes a block-oriented device

Format

proxyfile deviceitem filepath

Description

This command is used to provide "proxy" device driver services which allow the contents of a file to be accessed as if the file were a block-structured disk device capable of supporting a filesystem.

This command is NOT intended to be invoked directly from the shell. Rather, it's intended to be invoked as a server process by the proxy driver, when a proxied (virtual) device is opened. The proxy driver locates this program, and identifies the path to the file to be proxied, by examining entries in the DDF (Device Description File) for the proxied device.

A sample DDF which uses this server program might be:

```

version 1.0      // Note:  this is a per-file version.

driver proxyfile // this names the device visible to client
uses proxy

    needs
        nothing = 0
    end needs

    provides
        cmds: CMD_STATUS, CMD_BLOCKREAD,
              CMD_BLOCKWRITE, CMD_GETMAPINFO,
              CMD_MAPRANGE,  CMD_UNMAPRANGE,
              CMD_PREFER_FSTYPE
        SERVERPATH: "System.M2/Programs/ProxyFile"
        SERVERARG:  "/remote/mount.this.image"
        FS: 10
    end provides
end driver

```

If this DDF file is present in the System.M2/Devices directory of an "blessed" M2 filesystem, the DDF will be loaded into memory when the filesystem is mounted. Subsequently, any attempt to open the device named in this DDF will result in the proxy driver being loaded. The proxy driver will create the named device, and will create a task which runs the proxyfile program. The proxyfile program will then serve as a user-mode "device driver", making the contents of the "mount.this.image" file appear as the contents of the device.

One would normally use this device driver by issuing a shell command such as "mount proxyfile" or "mount foobar" or whatever. Note that the name of the proxied device (given in the "driver" line of the DDF) does not need to be the same as the name of the proxyfile program itself (given in the SERVERPATH line). A single copy of the server program "proxyfile" can be used to support any number of different proxied devices (each of which must of course have a unique name).

Implementation

Command implemented in V30.

Location

System.m2/Programs/ProxyFile

See Also

Mount

RecheckFS

Recheck all filesystems to see if they are still online.

Format

RecheckFS

Description

RecheckFS uses the `RecheckAllFileSystems()` call to ask the Portfolio filesystem manager to verify that all mounted filesystems are still on-line, and to start dismounting any which are not.

Implementation

Command implemented in V29.

Location

System.m2/Programs/RecheckFS

See Also

Mount, Dismount

Rename

Renames a file or directory.

Format

Rename <oldName> <newName>

Description

This command lets you change the name of a file or directory.

Argument

oldName

The current name of the file or directory to rename.

newName

The new name for the file or directory.

Implementation

Command implemented in V30.

Location

System.m2/Programs/Rename

RmDir

Removes directories.

Format`RmDir [-r] {directory}`**Description**

This command deletes directories. If -r is not specified, RmDir is unable to delete files and can only delete empty directories. If -r is specified, RmDir recursively deletes all files or directories under the specified argument list.

Arguments`[-r]`

Causes recursive deletion of any files or directories contained within the directories being deleted.

`{directory}`

Names of the directories to delete. If the -r option is not supplied, then these directories must be empty otherwise the delete operation will fail.

Implementation

Command implemented in V29.

Location

System.m2/Programs/RmDir

See Also

Delete, Mkdir

setalias

Set a file path alias.

Format

setalias <alias> <str>

Description

This command lets you create a filesystem path alias. Once created, the alias can be referenced from anywhere in the system.

Arguments

<alias>

The name of the alias to create.

<str>

The string that should be substituted whenever the alias is encountered when parsing directory and file names.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

alias

setbg

Set the shell's default behavior to background execution mode.

Format

setbg

Description

This command sets the shell's default execution mode to background. This prevents the shell from waiting for tasks to complete when they are executed. The shell returns immediately and is ready to accept more commands.

If the shell is currently in foreground mode and you wish to execute only a single program in background mode, you can append a '&' at the end of the command-line.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

bg

See Also

setfg

setcd

Set the shell's current directory.

Format

setcd [directory name]

Description

The shell maintains the concept of a current directory. Files within the current directory can be referenced without a full path specification, in a relative manner.

This command lets you specify the name of a directory that should become the current directory.

The current directory of the shell can be displayed using the the showcd command, or by executing setcd with no arguments.

Arguments

[directory name]

The name of the directory that should become the new current directory. If this argument is not supplied, then the name of the current directory is displayed, and is not changed.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

cd

See Also

showcd

setfg

Set the shell's default behavior to foreground execution mode.

Format

setfg

Description

This command sets the shell's default execution mode to foreground. This forces the shell to wait for tasks to complete when they are executed. The shell will not accept new commands until the current task completes.

If the shell is currently in background mode and you wish to execute only a single program in foreground mode, you can append a '#' at the end of the command-line.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

fg

See Also

setbg

setmaxmem

Set the amount of memory available in the system to the maximum amount possible.

Format

setmaxmem

Description

This command causes the shell to adjust the amount of memory available in the system to be the maximum supported by the hardware. This effectively undoes a previous use of the setminmem command.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

maxmem

See Also

setminmem

setminmem

Set the amount of memory available in the system to the minimum amount of memory guaranteed to be available in a production environment.

Format

setminmem

Description

This command causes the shell to adjust the amount of memory available in the system to match the minimum amount a memory a title must be able to run in.

The setmaxmem command can be used to restore the amount of memory to the maximum available in the current system.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

minmem

See Also

setmaxmem

setpri

Set the shell's priority.

Format

setpri [priority]

Description

This command sets the priority of the shell's task. It is sometimes desirable to boost the shell's priority to a high number. This lets commands such as showtask or memmap work with more accuracy.

The current priority of the shell can be displayed by entering this command with no argument.

Arguments

[priority]

The new shell priority. This value must be in the range 10..199. This number can be in decimal, or in hexadecimal starting with 0x or \$. If you don't specify a priority then the current priority is simply displayed.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

shellpri

showavailmem

Display information about the amount of memory currently available in the system.

Format

showavailmem

Description

This command displays information about the amount of memory installed in the system, and the amount of memory currently free.

This command also displays the amount of memory currently consumed by supervisor-mode code.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

availmem

See Also

showmemmap, showfreeblocks

showcd

Show the name of the current directory.

Format

showcd

Description

This command displays the name of the current directory to the debugging terminal.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

pcd

See Also

setcd

showerror

Display an error string associated with a system error code.

Format

showerr <error number>

Description

This command displays the error string associated with a numerical system error code.

Arguments

<error number>

The error code to display the string of. This number can be in decimal, or in hexadecimal starting with 0x or \$.

Implementation

Command implemented in shell V21.

Location

Built-in shell command.

Synonyms

err

showfreeblocks

Show the contents of a memory list.

Format

showfreeblocks [itemlname]

Description

This command displays all the chunks of memory currently available in the list of pages of a task, or of the whole system. The list of blocks displayed for the system includes all currently unallocated blocks of memory.

Arguments

[task name | task item number]

This specifies the name or item number of the task to display the data for. If this argument is not supplied, then the data for the system free list is displayed.

Implementation

Command implemented in shell V27.

Location

Built-in shell command.

See Also

showmemmap, showavailmem, showtask

showmemmap

Display a page map showing which pages of memory are used and free in the system, and which task owns which pages.

Format

showmemmap [task name | task item number]

Description

This command displays a page map on the debugging terminal showing all memory pages in the system, and which task owns each page.

Arguments

[task name | task item number]

This specifies the name or item number of the task to pay special attention to. The item number can be in decimal, or in hexadecimal starting with 0x or When a task is specified, any pages owned by that task will be marked with a '*'. This makes it quicker to find pages used by a specific task.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

memmap

See Also

showavailmem, showfreeblocks

showtask

Display information about tasks in the system.

Format

showtask [task name | task item number]

Description

This command displays information about task and threads in the system. It can also be given a specific task or thread name, in which case a more detailed output is produced describing the specified task exclusively.

Arguments

[task name | task item num]

This specifies the name or item number of the task to display the information about. The item number can be in decimal, or in hexadecimal starting with 0x or \$. If this argument is not supplied, then general information about all tasks in the system is displayed.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

Synonyms

ps

sleep

Cause the shell to pause for a number of seconds.

Format

sleep <number of seconds>

Description

This command tells the shell to go to sleep for a given number of seconds. The shell will not accept commands while it is sleeping.

Arguments

<number of seconds>

The number of seconds to sleep for. This number can be in decimal, or in hexadecimal starting with 0x or \$.

Implementation

Command implemented in shell V20.

Location

Built-in shell command.

SyncStress

Put some stress of task synchronization code to uncover title bugs.

Format

SyncStress [-quit]

Description

This program is used to test the strength of the task synchronization code in a title.

To use this program, simply invoke it from the shell. The program will setup conditions to cause random task switches to occur in the system. This will likely uncover any incorrect synchronization code that may exist in a title.

It is a bad idea to loosely synchronize the execution of multiple threads purely based on the task switching algorithm of the system. This generally causes the code to break if a task switch occurs at an unexpected moment. Proper sync methods should be used, including signals, messages, and semaphores.

Arguments

[-quit]

Instructs a previously invoked instance of the program to exit.

Implementation

Command implemented in V27.

Location

System.m2/Programs/SyncStress

Type

Types a file's content to the output terminal.

Synopsis

```
Type -hex {file}
```

Description

Reads a file and send its contents to the Debugger Terminal window.

Arguments

-hex

Display the contents of the file in hexadecimal form.

{file}

Names of the files to type. You can specify an arbitrary number of file names, and they will all get typed out.

Location

System.m2/Programs/Type

TypeIFF

Displays contents of an IFF file.

Format

TypeIFF [-short|-full] <file name>...

Description

This command displays the contents of the named IFF files to the debugging terminal. By default, just the chunk names, sizes, and hierarchy are displayed. Optional switches provide a way to display the contents of chunks in hexadecimal.

Arguments

<file name>

Name of an IFF file to display.

-short

Display the first 256 bytes of each chunk.

-full

Display the entire contents of each chunk.

Implementation

Command implemented in V27.

Location

System.m2/Programs/TypeIFF

Walker

Recursively displays the contents of a directory, and all nested directories.

Format

Walker {directory}

Description

This program demonstrates how to scan a directory, and recursively scan any directories it contains.

Arguments

{directory}

Names of the directories to list. You can specify an arbitrary number of directory names, and they will all get listed. If no name is specified, then the current directory gets displayed.

Location

System.m2/Programs/Walker

WriteMedia

Writes new raw data to media.

Format

```
WriteMedia [-verify]
           [-pattern <byte>]
           [-filename <media image file name>]
           <device name>
```

Description

This command writes raw data to a specified device. Either a constant value is written, or a file is used and its contents are written out directly to the media. Such a file would typically contain a file system image.

Arguments

- verify
Specifies that a verification pass is to be performed. This reads every byte of the card and makes sure it is set to the correct byte previously written to that location.
- pattern <byte>
Lets you specify the byte to use to fill the card. If this option is not supplied, 0 is used. This option is ignored if the -fileName option is used.
- fileName <media image file name>
Lets you specify the filename of an media image file. This file is written out directly to the target media. This option overrides the -pattern option. If the given file is not large enough to fill the whole media, the fill pattern will be used to fill out the remainder of the media.
- <device name>
Lets you specify the name of the device that contains the card to erase. This is typically "storagecard".

Location

System.m2/Programs/WriteMedia

audioavail

Show available audio resources in system.

Format

audioavail

Description

Prints available audio resources to the debug console.

Implementation

Also implemented as a command in V30.

Location

System.m2/Programs/audioavail

See Also

GetAudioResourceInfo()

dspfadrs

Try out DSP instruments or patches.

Format

```
dspfadrs [-2|-8] [-LineIn] [-Mute]
    <instrument name> [-sample <sample name>]
    [ <instrument name> [-sample <sample name>] ... ]
```

Description

This program is a diagnostic and experimentation tool to work with DSP instruments and patches. It loads the named DSP instruments or patches, and connects them in a chain by connecting the output port "Output" of one to the input port "Input" of the next. You may specify as many instrument or patch names as you wish. If the last instrument in the chain has an output named "Output", it is connected to `line_out.dsp`. If the first instrument in the chain has an input named "Input", a signal (either `noise.dsp` or `line_in.dsp`) is injected into this port.

If an instrument name is followed by a `-sample` argument, the program attempts to attach the named sample to the instrument.

The instrument is triggered using the 3DO control pad. Faders representing instrument knobs are displayed on the 3DO display, and can be controlled using the 3DO control pad.

Also displays tick benchmarking information about the instrument being tested:

Alloc

Number of ticks per frame that instrument allocates.

Cur

Number of ticks required in the current frame for the instrument to execute.

Max

Largest value of Cur since the program was started.

Arguments

<instrument name>

Name of a DSP instrument or binary patch file to demo (e.g., `svfilter.dsp`).

`-sample <sample name>`

Sample to connect to sample playing instruments (e.g., `sampler_16_v1.dsp`). If not specified, no sample is connected to the instrument. There is also no provision to ensure that the sample format and the sample player match. The sample is attached to the hook named 'InFIFO'.

`-2`

Run test instruments at 1/2 rate. Defaults to full rate.

`-8`

Run test instruments at 1/8 rate. Defaults to full rate.

`-LineIn`

By default, white noise (`noise.dsp`) is injected into the port named Input of the first test instrument. When this switch is used, `line_in.dsp` is injected instead. If the first test instrument has no port named Input, no signal is injected.

-Mute

Set Amplitude knob (if there is one) to 0 before starting instruments.

Controls

Fader Mode:

A

Toggle between Start and Release of instrument chain.

B

Start instrument chain when pressed, Release instrument chain when released (like a key on a MIDI keyboard).

C

Reset knobs to default values.

P

Enter oscilloscope mode.

X

Quit

Up, Down

Select a fader by moving up or down.

Left, Right

Adjust current fader (coarse).

LShift + Left, Right

Adjust current fader (fine).

LShift + Up, Down

Switch to previous/next instrument fader screen.

Oscilloscope Mode:

A

Capture new sample into oscilloscope.

C, X

Return to fader mode.

Up, Down

Increase, decrease vertical magnification of oscilloscope.

Left, Right

Decrease, increase horizontal magnification of oscilloscope.

LShift, RShift

Scroll oscilloscope left, right.

Implementation

Command implemented in V24.

Location

System.m2/Programs/dspfaders

See Also

makepatch

insinfo

Displays information about DSP Instruments.

Format`insinfo [<instrument name> | <item number>]...`**Description**

This command displays information about the requested instruments. The information is sent out to the debugging terminal.

Arguments

<instrument name>

Name of a DSP instrument or patch file to display.

<item number>

The item number of an existing Instrument or Template item to display. The item number can be in decimal, or in hexadecimal starting with 0x.

Implementation

Command implemented in V30.

Location

System.m2/Programs/insinfo

See Also

`GetInstrumentResourceInfo()`,
`GetAttachments()`

`GetInstrumentPortInfoByName()`,

makepatch

Reads patch script language, writes binary patch file.

Format

```
makepatch <source script> <output file> [-check]
```

Description

Reads a patch script language to construct a binary patch file of the same format as written by ARIA. Patches created in this way can be loaded with `LoadPatchTemplate()`, added to a `PIMap`, etc.

You can test your newly created patch with `dspfaders`.

Arguments

<source script>

File name of patch script to read.

<output file>

File name of output patch file to write. If an error occurs during parsing or writing of the file, the output file is deleted. Append the suffix `.patch` to your output patch name if you want `LoadScoreTemplate()` to be able to load it (e.g., `foo.patch`).

-check

Loads the output patch file to test its validity. Note that the patch file is not deleted when an error is detected while loading.

Patch Script Language

The patch script language is line-oriented (i.e., one command per line; one line per command). Each line consists of white space separated words, of which the first word is the command. Leading white space is ignored. All commands, keywords, and switches are matched case-insensitively. All text following a semicolon (;) or pound sign (#) to the end of the line is ignored. Blank lines are ignored.

With the noted exceptions, commands may appear in any order.

Instrument Blocks:

The following commands each add an instrument block with the specified block name to the patch. Block names are matched case-insensitively, and must be unique. Multiple blocks from the same template may be added to the patch. Each of these commands results in a `PATCH_CMD_ADD_TEMPLATE`.

Instrument <block name> <template file name>

Adds an instrument template block with the name <block name> to the patch by loading the named standard DSP Instrument Template (e.g., `sawtooth.dsp`).

Mixer <block name> <num inputs> <num outputs> [-LineOut] [-Amplitude]

Adds an instrument template block with the name <block name> to the patch by creating a Mixer of the specified configuration. -LineOut and -Amplitude correspond to the Mixer flags `AF_F_MIXER_WITH_LINE_OUT` and `AF_F_MIXER_WITH_AMPLITUDE`, respectively. See `CreateMixerTemplate()` for more detail.

Ports:

The commands in this group create patch ports (e.g., knobs, inputs, and outputs) or define internal connections

between blocks and ports.

Connect <from block name> <from port name> <from part num> <to block name> <to port name> <to part num>

Creates an internal connection between a pair of constituent template (block) ports, patch inputs, patch outputs, or patch knobs. Use a period (.) for the block name when referring to one of the patch's inputs, knobs, or outputs (see the example below). This results in a PATCH_CMD_CONNECT.

Constant <block name> <port name> <part num> <value>

Assigns a constant value to a constituent template (block) input or knob. This results in a PATCH_CMD_SET_CONSTANT.

Expose <exposed port name> <source block name> <source port name>

Exposes a FIFO, Envelope Hook, or Trigger of one of the patch's constituent templates (blocks) to make it accessible to clients of the patch. This results in a PATCH_CMD_EXPOSE.

Input <port name> <num parts> <signal type>

Define an input port for the patch. This results in a PATCH_CMD_DEFINE_PORT of AF_PORT_TYPE_INPUT.

Knob <knob name> <num parts> <signal type> <default>

Define a knob for the patch. This results in a PATCH_CMD_DEFINE_KNOB.

Output <port name> <num parts> <signal type>

Define an output port for the patch. This results in a PATCH_CMD_DEFINE_PORT of AF_PORT_TYPE_OUTPUT.

Attachments:

The commands in this group create slave Items (e.g., samples and envelopes) and their attachments to the patch. Each slave item is given a case-insensitively matched, unique item name.

Attach <item name> <exposed port name> [-StartAt <offset>] [-FatLadySings]

Attaches a previously defined slave item (sample, delay line, or envelope) identified by <item name> to the exposed FIFO or Envelope port named <exposed port name>.

-StartAt <offset> permits setting a start offset (sample frame number or envelope segment index) for the attachment. This is particularly useful for setting delay times for delay line attachments. This corresponds to the Attachment tag AF_TAG_START_AT.

-FatLadySings indicates that the instrument should stop when this attachment completes. This is handy for envelope attachments. This corresponds to the Attachment flag AF_ATT_FATLADYSINGS.

Note: This command must appear after the definition of the item to be attached.

Note: The Attach command can only attach to exposed ports. See the Expose command.

DelayLine <item name> <num bytes> <num channels> [-NoLoop]

Creates a delay line. See CreateDelayLine() for explanation of the arguments.

Note: like other attached items, delay lines belong to the patch template, so all instruments

created from such a patch share the same delay line. This fact cripples the delay line feature for use in any application requiring the creation of multiple instruments from a patch. This feature may still be used in cases where only one instrument is to be allocated from the patch.

If you need a delay line per instrument, create the patch with the delay related instruments but without the delay line itself. When you create instruments from the patch, create and attach the delay line yourself. You may make use of the attachment tag `AF_TAG_AUTO_DELETE_SLAVE` to simplify cleanup of the instrument delay lines.

`Envelope <item name> <signal type> [-LockTimeScale] [-PitchTimeScaling <base note> <notes/octave>] <value 0> <duration 0> ... <value N-1> <duration N-1>`

Creates an `Envelope` with `N` `EnvelopeSegments`. Values are expressed in the appropriate units for the specified signal type. Duration is expressed in seconds.

`-LockTimeScale` causes the `AF_ENVF_LOCKTIMESCALE` flag to be set for the `Envelope`. This prevents the `StartInstrument()` and `ReleaseInstrument()` tag `AF_TAG_TIME_SCALE_FP` from having any effect on this envelope.

`-PitchTimeScaling` sets pitch-based time scaling parameters for the envelope. `<base note>` is the MIDI note number at which time scaling factor is 1.0. This defaults to 60 (middle C). `<notes/octave>` is the number of semitones at which pitch time scale doubles. A positive value makes the envelope shorter as pitch increases; a negative value makes the envelope longer as pitch increases. Zero (the default) disables pitch-based time scaling. This corresponds to the `Envelope` tags `AF_TAG_BASENOTE` and `AF_TAG_NOTESPEROCTAVE`.

Envelopes are created without loop points. To set these, apply the `EnvSustainLoop`, `EnvReleaseLoop`, and `EnvReleaseJump` commands after creating the envelope.

`EnvReleaseJump <item name> <to segment>`

Sets the release jump point for a previously defined envelope. This is the segment in the envelope to jump to when the instrument is released. This corresponds to the `Envelope` tag `AF_TAG_RELEASEJUMP`.

`EnvReleaseLoop <item name> <begin segment> <end segment> <loop time>`

Sets the release loop for a previously defined envelope. `<begin segment>` and `<end segment>` are indices in the envelope's `EnvelopeSegment` array. `<loop time>` is the duration in seconds for the segment described by looping from `<end segment>` back to `<begin segment>`.

See the documentation for the `Envelope` tags `AF_TAG_RELEASEBEGIN`, `AF_TAG_RELEASEEND`, and `AF_TAG_RELEASETIME_FP` for more detail.

`EnvSustainLoop <item name> <begin segment> <end segment> <loop time>`

Sets the sustain loop for a previously defined envelope. `<begin segment>` and `<end segment>` are indices in the envelope's `EnvelopeSegment` array. `<loop time>` is the duration in seconds for the segment described by looping from `<end segment>` back to `<begin segment>`.

See the documentation for the `Envelope` tags `AF_TAG_SUSTAINBEGIN`, `AF_TAG_SUSTAINEND`, and `AF_TAG_SUSTAINTIME_FP` for more detail.

`Sample <item name> <sample file name>`

Loads an AIFF sample file to embed in patch file.

Compiler Options:

These commands affect the compilation process.

Coherence [On|Off]

When set to On (the default state), the patch is to be built in such a way as to guarantee signal phase coherence along all internal connections. When set to Off, signals output from one constituent instrument may not propagate into the destination constituent instrument until the next audio frame. The resulting patch Template is slightly smaller when Coherence is Off.

See PATCH_CMD_SET_COHERENCE for more details.

Signal Types:

The following names are recognized for the <signal type> argument of the commands described above. They are named similarly to the AF_SIGNAL_TYPE_ constants defined in <:audio:audio.h>.

Signed

Signed signal values in the range of 1.0..1.0.

Unsigned

Unsigned signal values in the range of 0.0..2.0.

OscFreq

Oscillator frequency values in the range of -22050.0..22050.0 Hz.

LFOFreq

LFO frequency values in the range of -86.1..86.1 Hz.

SampRate

Sample rate values in the range of 0.0..88100.0 Hz.

Whole

Integer values in the range of -32768..32767.

Implementation

Command in V29.

Location

System.m2/Programs/makepatch

Caveats

See the DelayLine command description above for a serious limitation of its use.

The patch script parser only enforces syntax, not content. Unless the -check option is specified, makepatch will happily write a bogus patch file.

No support for custom tunings or nested patches yet.

Examples


```
; sawtooth oscillator with amplitude envelope

; instruments
Instrument env envelope.dsp
Instrument osc sawtooth.dsp

; ports and knobs
Knob Frequency 1 OscFreq 261.63 ; default to middle C
Knob Amplitude 1 Signed 1.0      ; default to full amplitude
Output Output 1 Signed

; expose so we can attach envelope below
Expose AmpEnv env Env

; route patch's Frequency and Amplitude knobs
Connect . Frequency 0 osc Frequency 0
Connect . Amplitude 0 env Amplitude 0

; connect envelope output to oscillator amplitude
Connect env Output 0 osc Amplitude 0

; connect oscillator output to patch's output
Connect osc Output 0 . Output 0

; envelope
;
;           A           D       S       R
Envelope env Signed  0 1.0  1.0 0.5  0.5 0.2  0 0
EnvSustainLoop env 2 2 0.0      ; sustain at single point.
EnvReleaseJump env 2           ; jump immediately to release, even
if not
                                ; at sustain yet.
Attach env AmpEnv -FatLadySings ; -FatLadySings causes instrument to
stop
                                ; when envelope completes.
```

See Also

PatchCmd, --Patch-File-Overview--, LoadPatchTemplate(), dspfaders,

playmf

Plays a standard MIDI file.

Format

```
playmf <MIDI file> <PIMap file> [-n<NumReps>] [-a<Amplitude>] [-verbose]
```

Description

Loads a standard MIDI format file, loads instruments and AIFF samples described in a PIMap file, and plays the MIDI file the specified number of times. Demonstrates use of the Juggler and the score playing routines.

This program is implemented as both a shell program (in System.m2/Programs) and as an example.

Arguments

<MIDI file>

File name of a Format 0 or Format 1 MIDI file.

<PIMap file>

File name of a PIMap (Program-Instrument Map) text file. This file is parsed at run-time to associate MIDI program numbers with audio folio Instrument Templates, Samples, etc.

-n<NumReps>

NumReps is number of times to play the MIDI file. Defaults to 1. For example: -n10 for 10 repetitions.

-a<Amplitude>

Amplitude per voice that is passed to `CreateScoreMixer()`. The value is expressed as a percentage of maximum. Defaults to 12. For example: -a40 for louder music. Warning if you set this too high you will cause clipping of the audio.

-verbose

When specified, causes numerous informational messages to be printed.

Implementation

Released as an example in V20.

Also implemented as a command in V24.

Location

System.m2/Programs/playmf, Examples/Audio/Score

See Also

PIMap, minmax_audio.c

MPEG Video Decompression Device

This chapter describes the operation, benefits, and limitations of the 3DO Portfolio MPEG video decompression device, which uses a 3DO-designed video decoder.

The information presented here is intended for programmers who want to use MPEG video with the M2, and assumes that you are familiar with the MPEG-1 video standard.

Note: *The term MPEG will be used throughout this document to refer to the MPEG-1 audio/video coding standard.*

Most title developers should avoid using the MPEG device interface directly, and should instead use the 3DO DataStreamer library MPEG Video Subscriber interface.

This chapter contains the following topics:

Topic	Page
Using MPEG in a 3DO Title	140
Using the Portfolio MPEG Device	141
Video Decoding Options	151
MPEG Still Image Decoding	155

For further information on MPEG, see the following documents:

- ◆ ISO/IEC 11172-1 MPEG-1 Systems (standard multiplex format used for video CD, but not for interactive M2 titles)
- ◆ ISO/IEC 11172-2 MPEG-1 Video (standard video compression format)

Using MPEG in a 3DO Title

MPEG is easy to use and improves the performance of almost any 3DO title. In this section, we'll cover the following topics:

- ◆ Benefits of MPEG
- ◆ Typical Data Flow in the Portfolio environment
- ◆ Video Synchronization

Benefits of MPEG

MPEG provides significant compression of video data without the marked loss in quality associated with other compression methods, such as Cinepak. Decompressed MPEG video is comparable in quality to that played from VHS tape.

MPEG video is typically compressed to 1.15 Mbits/s and decompressed to 320 x 240 x 24 bits/pixel at 30 frames per second, resulting in a compression ratio of more than 50:1.

This high compression permits full-screen, full-motion video using the single-speed CD bandwidth of 150 KB/s and allows for up to 74 minutes of playback from a single CD.

In the 3DO system, full-screen, full-motion video can be played back using only half of the double-speed CD drive's 300 KB/s bandwidth, with additional data interleaved with the MPEG data. However, you can specify the bandwidth of the compressed stream to be greater or less than 1.15 Mbits/s, allowing you to fine-tune the trade-off between video quality, bandwidth, and stream size. Picture size and frame rate are also programmable.

Typical Data Flow

In the Portfolio environment, MPEG data typically flows through the 3DO system in the following manner:

1. The application reads encoded MPEG data from the CD-ROM and places it into buffers in memory.
2. The application demultiplexes the buffers (if necessary) and sends the encoded data to the decoder device via write I/O requests.
3. The device decodes the data and places the results in frame buffers provided by the application via read I/O requests.
4. At the appropriate time, the application presents (displays) the decoded data by page-flipping to the decoded frame buffers, or by composing the decoded frames into other frame buffers.
5. Data buffers are recirculated after being used.

It is important that the application manage its buffers carefully to prevent underflow or overflow during normal play. Allowing encoded data buffers to underflow may create glitches in the video output. Overflowing the buffers may require additional buffering to cover the rotational latency caused by the inability to continuously stream data from the CD.

Video Synchronization

Presenting decoded video data at the appropriate rate and time is the responsibility of the library application. To aid in synchronization, the MPEG device supports tagging MPEG video presentation units (frames) with presentation time stamps (PTS).

Note: *If the term PTS is unfamiliar to you, consult the MPEG-1 Systems document.*

Time stamps are attached to the encoded data's write I/O requests. Each time stamp propagates through the decoder pipeline and is returned attached to the appropriate read I/O request. The application then uses the time stamp information to determine when to present the decoded data. Usually, the application presents the video data when the system clock is greater than or equal to the corresponding video time stamp.

The data preparation tools, in concert with the application, are responsible for interleaving and demultiplexing the video data in such a way that the decoder can decode the data at the appropriate time without underflowing or overflowing the application's buffers.

Using the Portfolio MPEG Device

The MPEG video device conforms to the Portfolio I/O device standard interface and responds to specialized MPEG versions of the read, write, status, and abort commands. The control command permits additional operations. This section describes in detail how to use the MPEG device. Much of this information can also be found in the Portfolio documentation under the section "Handling I/O."

The non-standard macros, constants, and data types used in the following examples are defined in the interface file `<device/mpegvideo.h>` and `<kernel/devicecmd.h.>`

Note: *As always, the return values from any system call must be checked for errors. The following examples do not show error checking for the sake of brevity and clarity.*

Common Operations

This section lists the operations required to use the MPEG decoder.

Opening the Device

The device must be opened before any operations can be performed (see Example 1-1).

Example 1-1 *Opening the MPEG device.*

```
/* Open the MPEG video device */
err = CreateDeviceStackListVA(&list, "cmds", DDF_EQ,
                              DDF_INT, 1,
                              MPEGVIDEOCMD_CONTROL,
                              NULL);

if (err < 0)
{
    printf("cannot create device stack list\n");
    exit( 1 );
}
if ( IsEmptyList(list) )
{
    printf("no MPEG devices found\n");
    exit( 1 );
}
videoDevice = OpenDeviceStack((DeviceStack*)
                              FirstNode(list));
DeleteDeviceStackList(list);
```

Creating IOReq Structures

As with any Portfolio device, requests are made of the device by using IOReqs. An application typically uses one IOReq per buffer and an additional IOReq for commands (see Example 1-2).

In the following example, we assume that the application will be notified of I/O completion by signal. It is also possible to be notified by a message reply. See the Portfolio documentation under the section "Handling I/O."

Example 1-2 *Creating I/O requests.*

```
...
Item myIOReq[ NUM_BUFFERS ];
Item myCommandReq;

for( i = 0; i < NUM_BUFFERS; i++ )
{
    myIOReq[ i ] = CreateIOReq( 0,0,mpegDevice,0 );
}
myCommandReq = CreateIOReq( 0,0,videoDevice,0 );
...
```

Allocating Buffers

Write buffers containing compressed MPEG data can be allocated anywhere in memory with no alignment restrictions.

Read buffers must be allocated as shown in Example 1-3 and attached to a bitmap item. The bitmap item is then passed to the MPEG video device in the read I/O request.

Often, MPEG write buffers are not allocated separately for each IOReq. For example, several large disc read buffers may be allocated for reading in a multiplexed stream from the CD. Then, after parsing the multiplexed stream, pointers to the MPEG data within these write buffers are passed to the device through the IOReqs.

Example 1-3 *Allocating read buffers.*

```
Item allocbitmap (int32 wide, int32 high, int32 type)
{
    Bitmap *bm;
    Item item;
    Err err;
    void *buf;

    item = CreateItemVA (MKNOEID (NST_GRAPHICS,
                                GFX_BITMAP_NODE),
                        BMTAG_WIDTH, wide,
                        BMTAG_HEIGHT, high
                        BMTAG_TYPE, type
                        BMTAG_DISPLAYABLE, TRUE
                        BMTAG_MPEGABLE, TRUE
                        TAG_END);

    if (item<0)
    {
        PrintfSysErr (item);
        die ("Bitmap creation failed. \n");
    }
    bm = LookupItem (item);

    buf = AllocMemMasked (bm->bm_BufferSize,
                        (bm->bm_BufMemType,
                        (bm->bm_BufMemCareBits,
                        (bm->bm_BufMemStateBits,

    if (!buf)
        die ("Can't allocate bitmap buffer. \n");

    memset (buf, 0x800000L, bm->bm_BufferSize);

    if ((err = ModifyGraphicsItemVA (item, BMTAG_BUFFER, buf,
                                    TAG_END))<0)
    {
        PrintfSysErr (err);
        die ("Can't modify Bitmap Item.\n");
    }
    return (item);
}
```

Creating IOInfo Structures

As with any Portfolio device, the application uses IOInfo structures to pass information to the device, typically one structure per IOReq. All used fields in the IOInfo structure must be set and *all unused fields MUST be set to 0*. One IOInfo struct may suffice for your application because SendIO copies its contents into the IOReq.

Sending Read and Write Requests

All read and write requests should be sent to the device via SendIO(). The device can queue any number of read and write requests and completes them asynchronously. Attempting to send read or write requests using DoIO() will likely result in deadlock (see the "Avoiding Deadlock" section). Read and write requests are shown in Examples 7-4 and 7-5 respectively.

Example 1-4 Read request

```
/* sending a read request to the MPEG video device */
Item bitmapItem, readReqItem;
Bitmap *bitmap;
IOInfo readInfo;

...
/* allocate bitmap, create IOReqs, etc... */
...

memset( &readInfo, 0, sizeof( IOInfo ) );

readInfo.ioi_Command = MPEGVIDEOCMD_READ;
readInfo.ioi_Send.iob_Buffer = &bitmapItem;
readInfo.ioi_Send.iob_Len = sizeof( Item );

SendIO( readReqItem, &readInfo);
```

Example 1-5 Write request

```
/* sending a write request to the MPEG video device */
Item writeReqItem;
IOInfo writeInfo;
uint8 writeBuffer[ WRITEBUFFERSIZE ];

...
/* allocate and fill write buffer, create IOReqs, etc */
...
memset( &writeInfo, 0, sizeof( IOInfo ) );

writeInfo.ioi_Command = MPEGVIDEOCMD_WRITE;
writeInfo.ioi_Send.iob_Buffer = writeBuffer;
writeInfo.ioi_Send.iob_Len = sizeof( writeBuffer );

SendIO( writeReqItem, &writeInfo);
```

Waiting for Completion

The application determines completion of a read or write request via a signal or message reply. Example 1-6 assumes that the application waits for a signal. Your application should process completed IOReqs in FIFO order, and then stop at the first incomplete IOReq.

It is important that the application call `WaitSignal()` and not just busy wait on `CheckIO()`. Busy waiting wastes CPU cycles. Also, the application must call `WaitSignal(SIGF_IODONE)` so that it can respond to the completion of both read and write requests, otherwise deadlock may result.

Example 1-6 *Waiting for the completion of I/O Request*

```
int32 waitResult;

while( not_done )
{
    /* wait for a signal that an I/O request has completed */
    waitResult = WaitSignal( SIGF_IODONE );

    /* handle errors / aborts */
    ...

    /* check each read I/O request to see if it has completed
    */
    for( i = queued IOReq numbers in FIFO order)
    {
        if( CheckIO( myIOReq[ i ] ) )
        {
            /* refill completed write buffers, or
            use completed read buffers */
            ...

            /* queue the new request, (this may be done later, */
            /* after the buffer has been used or refilled */
            SendIO( myIOReq[ i ], & myIOI[ i ] );
        }
        else break;
    }
    /* check each write I/O request to see if it has completed
    */
}
```

Avoiding Deadlock

The MPEG decoder pipelines are decompression FIFOs: compressed data enters the decoders through write requests, and decompressed data is output through read requests. If there is no input data, the output stops, and if there are no output buffers queued, the decoders stop processing data. If the application does not handle these operations correctly, deadlock can result. The following example can result in deadlock:

Example 1-7 *Example of code that may result in deadlock.*

```
...
/* open device, create ioreqs etc.. */
...

SendIO( myIOReq[ 0 ], & myIOI[ 0 ] );
WaitIO( myIOReq[ 0 ] );
...
```

If the sent IOReq is a read request and the decoder does not have enough compressed data queued to decode a frame into the read buffer, the IOReq will NEVER complete. The application must maintain a continuous flow of encoded data buffers and read buffers to the device to prevent deadlock. There also is no magic ratio of number of bytes in to number of bytes out that allows any application to assume that a certain number of write requests will satisfy a certain number of read requests.

Sending Control Requests

Control requests are completed immediately (although a control request may cause future effects) and can be sent using DoIO(). Arguments for control are passed in the tag arguments of the CodecDeviceStatus structure. Example 1-8 sets the video output size to 320 x 240 in 16-bit mode.

Example 1-8 *Setting the video output size.*

```

...
IOInfo commandIOI;
CODECDeviceStatus myStatus;

memset( &commandIOI, 0, sizeof( IOInfo ) );
memset( &myStatus, 0, sizeof( CODECDeviceStatus ) );

myStatus.codec_TagArg[ 0 ].ta_Tag = VID_CODEC_TAG_HSIZE;
myStatus.codec_TagArg[ 0 ].ta_Arg = 320;
myStatus.codec_TagArg[ 1 ].ta_Tag = VID_CODEC_TAG_VSIZE;
myStatus.codec_TagArg[ 1 ].ta_Arg = 240;
myStatus.codec_TagArg[ 2 ].ta_Tag = VID_CODEC_TAG_DEPTH;
myStatus.codec_TagArg[ 2 ].ta_Arg = 16;
myStatus.codec_TagArg[ 3 ].ta_Tag = VID_CODEC_TAG_M2MODE;
myStatus.codec_TagArg[ 4 ].ta_Tag = TAG_END;

commandIOI.ioi_Command = MPEGVIDEOCMD_CONTROL;
commandIOI.ioi_Send.iob_Buffer = &myStatus;
commandIOI.ioi_Send.iob_Len = sizeof( CODECDeviceStatus );

DoIO( myCommandReq, &commandIOI );
...

```

Adding Synchronization

It is the responsibility of the application to present decoded video at the appropriate rate and time. To aid in this, the MPEG device supports passing presentation time stamps (PTSs) along with encoded data. The PTSs pass through the decoding pipeline and are output along with the corresponding decoded data. This must be done to account for the reordering of pictures in MPEG video and any decoder latency.

PTSs are normally generated when the video data is initially encoded and are placed in either the packet layer of an MPEG system multiplexed stream (see the MPEG documents if any of this is unfamiliar), or in the DataStreamer chunks. When the application demultiplexes the stream, it must parse out the PTSs and pass them to the device along with the encoded-video write request. The device then associates the PTS with the first Access Unit (compressed frame) which starts in the buffer pointed to by the write request.

Example 1-9 demonstrates how the application passes PTSs to the device using write requests. The application can also pass a 32-bit value along with the PTS.

Note: *In the MPEG-1 standard, PTSs are 33-bit values. The 33rd bit is currently ignored by the device.*

Example 1-9 *Passing PTSs to the device.*

```
...
FMVIOReqOptions myOptions[ NUM_BUFFERS ];
uint32 thePTS;

/* do all initial stuff, including parsing the PTS */
...

/* set all fields to zero */
memset( &myOptions[ i ], 0, sizeof( FMVIOReqOptions ) );

myOptions[ i ].FMVOpt_Flags = FMVValidPTS;
myOptions[ i ].FMVOpt_PTS = thePTS;

/* set up IOInfo (see write example) */
...
myIOI[ i ].ioi_Cmd = MPEGVIDEOCMD_WRITE;
...
myIOI[ i ].ioi_CmdOptions = (uint32 ) & myOptions[ i ] ;

SendIO( myIOReq[ i ], & myIOI[ i ] );
...
```

Note: *The myOptions memory must remain allocated and available to the device until the IOReq completes.*

There may be cases where some write requests will not have a corresponding PTS, while others do. If a write request does not have a PTS, the FMVValidPTS bit of the FMVOpt_Flags field should be set to zero.

The PTSs are returned to the application in the io_Extension fields of the read IOReq structure and can be retrieved using macros defined in the interface header file. Example 1-10 shows how this is performed:

Example 1-10 *Retrieving returned PTSs.*

```
...
IOReq *readIOReq;
uint32 thePTS, offset;

/* at this point a read request has completed */

readIOReq = (IOReq *) LookupItem( myIOReq[ i ] );
if( readIOReq->io_Flags & FMVValidPTS )
{
    thePTS = FMVGetPTSValue( readIOReq );
    offset = FMVGetPTSOffset( readIOReq );
}
...
```

Once the PTS values have been retrieved, the application presents the video data when the system clock value exceeds the picture's PTS value.

Video Decoding Options

The MPEG-1 video standard allows a wide range of possible picture sizes, frame rates, and compression. The Portfolio MPEG video device supports the parameters necessary for playing Whitebook (VideoCD) titles and more, including higher data rates (for better quality). M2 native 32- and 16-bit pixel output modes are supported.

Output Modes

This section describes the various video output modes and gives examples of how to set them. Note that horizontal resampling is not available on M2.

16-Bit Square Pixel Mode

This mode is useful for 'interactive' titles which may use the graphics engines to map decoded MPEG pictures onto some surface. On M2 systems, the decoded pixels are in M2 native 16-bit format suitable for use by the triangle engine. This mode can be set as shown in Example 1-11.

Example 1-11 *Setting 16-bit square pixel output mode.*

```
...
IOInfo commandIOI;
CODECDeviceStatus myStatus;

memset( &commandIOI, 0, sizeof( IOInfo ) );
memset( &myStatus, 0, sizeof( CODECDeviceStatus ) );

myStatus.codec_TagArg[ 0 ].ta_Tag = VID_CODEC_TAG_HSIZE;
myStatus.codec_TagArg[ 0 ].ta_Arg = 320;
myStatus.codec_TagArg[ 1 ].ta_Tag = VID_CODEC_TAG_VSIZE;
myStatus.codec_TagArg[ 1 ].ta_Arg = 240;
myStatus.codec_TagArg[ 2 ].ta_Tag = VID_CODEC_TAG_DEPTH;
myStatus.codec_TagArg[ 2 ].ta_Arg = 16;
myStatus.codec_TagArg[ 3 ].ta_Tag = VID_CODEC_TAG_M2MODE;
myStatus.codec_TagArg[ 4 ].ta_Tag = TAG_END;

commandIOI.ioi_Command = MPEGVIDEOCMD_CONTROL;
commandIOI.ioi_Send.iob_Buffer = &myStatus;
commandIOI.ioi_Send.iob_Len = sizeof( CODECDeviceStatus );

DoIO( myCommandReq, &commandIOI );
...
```

The size of the video read buffers (decompressed frame buffers) should be the dimensions of the encoded stream x 2 bytes and must be aligned to a 32-byte boundary. The horizontal and vertical dimensions must be multiples of 16 pixels.

24-Bit Square Pixel Mode

This mode is useful when the highest quality video is desired. The decoded pixels are in M2 native 32-bit format. This mode can be set as shown in Example 1-12.

The size of the video read buffers should be the dimensions of the picture x 4 bytes and must be aligned to a 32-byte boundary. It is possible to specify picture sizes smaller or larger than this.

The horizontal and vertical dimensions must be multiples of 16 pixels. In some situations (for example, if a Video CD player is used), the application must resample the video using the triangle engine.

Example 1-12 *Setting 24-bit square pixel output mode.*

```
...
IOInfo commandIOI;
CODECDeviceStatus myStatus;

memset( &commandIOI, 0, sizeof( IOInfo ) );
memset( &myStatus, 0, sizeof( CODECDeviceStatus ) );

myStatus.codec_TagArg[ 0 ].ta_Tag = VID_CODEC_TAG_HSIZE;
myStatus.codec_TagArg[ 0 ].ta_Arg = 320;
myStatus.codec_TagArg[ 1 ].ta_Tag = VID_CODEC_TAG_VSIZE;
myStatus.codec_TagArg[ 1 ].ta_Arg = 240;
myStatus.codec_TagArg[ 2 ].ta_Tag = VID_CODEC_TAG_DEPTH;
myStatus.codec_TagArg[ 2 ].ta_Arg = 24;
myStatus.codec_TagArg[ 3 ].ta_Tag = VID_CODEC_TAG_M2MODE;
myStatus.codec_TagArg[ 4 ].ta_Tag = TAG_END;

commandIOI.ioi_Command = MPEGVIDEOCMD_CONTROL;
commandIOI.ioi_Send.iob_Buffer = &myStatus;
commandIOI.ioi_Send.iob_Len = sizeof( CODECDeviceStatus );

DoIO( myCommandReq, &commandIOI );
...
```

Other Commands

The following commands are used to aid in synchronization and for special effects.

Scan Mode

To aid in the visual search of a video stream, such as in fast forward and fast reverse modes on Whitebook disks, the application can ask the video device to only decode I pictures.

In this way, the application can skip around in the video stream and occasionally display a new picture without generating corrupted pictures. Once set in this mode, the device will only decode I pictures until it receives a "play" command. This mode is set as shown in Example 1-13.

Example 1-13 Setting Scan Mode.

```
...
IOInfo commandIOI;
CODECDeviceStatus myStatus;

memset( &commandIOI, 0, sizeof( IOInfo ) );
memset( &myStatus, 0, sizeof( CODECDeviceStatus ) );

myStatus.codec_TagArg[ 0 ].ta_Tag = VID_CODEC_TAG_KEYFRAMES;
myStatus.codec_TagArg[ 1 ].ta_Tag = TAG_END;

commandIOI.ioi_Command = MPEGVIDEOCMD_CONTROL;
commandIOI.ioi_Send.iob_Buffer = &myStatus;
commandIOI.ioi_Send.iob_Len = sizeof( CODECDeviceStatus );

DoIO( myCommandReq, &commandIOI );
...
```

Play Mode

This command takes the video device out of Scan mode. This mode is set as shown in Example 1-14.

Example 1-14 Setting Play Mode.

```
...
IOInfo commandIOI;
CODECDeviceStatus myStatus;

memset( &commandIOI, 0, sizeof( IOInfo ) );
memset( &myStatus, 0, sizeof( CODECDeviceStatus ) );

myStatus.codec_TagArg[ 0 ].ta_Tag = VID_CODEC_TAG_PLAY;
myStatus.codec_TagArg[ 1 ].ta_Tag = TAG_END;

commandIOI.ioi_Command = MPEGVIDEOCMD_CONTROL;
commandIOI.ioi_Send.iob_Buffer = &myStatus;
commandIOI.ioi_Send.iob_Len = sizeof( CODECDeviceStatus );

DoIO( myCommandReq, &commandIOI );
...
```

MPEG Still Image Decoding

The MPEG device driver and hardware decoder can decompress single MPEG frames very efficiently. This is useful for decompressing images and textures. The images should be encoded as MPEG I (Intra) frames. Use the “Sparkle” application to convert any PICT files to MPEG still images. See the “3DO M2 Video Processing Guide” for more information about Sparkle.

To decode a single MPEG I frame into a bitmap, a special command (VID_CODEC_TAG_KEYFRAME) is sent to the MPEG device driver.

```
CODECDeviceStatus stat;
Item videoCmdItem;
IOInfo videoCmdInfo;

videoCmdItem = CreateIOReq(0, 0, mpegDevice, 0);
memset(&stat, 0, sizeof(CODECDeviceStatus));
...                               /* Other commands to the driver */
stat.codec.TagArg[i] = VID_CODEC_TAG_KEYFRAME;
                               /* Decode ONLY I Frames */
....
stat.codec.TagArg[i] = TAG_END;
videoCmdInfo.ioi_Command = MPEGVIDEOCMD_CONTROL;
videoCmdInfo.ioi_Send.iob_Buffer = &stat;
videoCmdInfo.ioi_Send.iob_Len = sizeof(CODECDeviceStatus);
err = DoIO(videoCmdItem, &videoCmdInfo);
```

In VID_CODEC_TAG_KEYFRAME mode, the MPEG device driver only looks for I frames to decode - it skips B and P frames.

MPEG Device Driver Memory Allocation

When the MPEG device driver decodes MPEG P (Predicted) and B (Bidirectional) frames, it allocates two internal buffers called *reference frame* buffers. The size of these buffers depends on the size of the MPEG movie specified in the MPEG video sequence header (part of standard MPEG stream syntax).

```
size = width * height * 1.5 bytes;
```

Opening the MPEG device driver does not allocate these buffers. The device driver defers allocation of these buffers until it reaches a MPEG video sequence header.

Whether you have an I frame-only movie, or just want to decode MPEG still images, the MPEG device driver does not need reference buffers and can save you a lot of memory. Prevent the device driver from allocating reference buffers by sending a VID_CODEC_TAG_KEYFRAME command to the device driver after opening the device and before sending any MPEG stream data.

To switch back to normal play mode, use the `VID_CODEC_TAG_PLAY` command.
For example code look at the *photo* and *playfast* example application sources.



3DO M2 Lonely Chapters

Version 2.0 – May 1996

Copyright © 1996 The 3DO Company and its licensors.

All rights reserved. This material constitutes confidential and proprietary information of The 3DO Company. This documentation is subject to a license agreement with The 3DO Company and may be used only by parties to such agreement. Use by any other persons, and/or for any purpose not expressly authorized by the agreement, is strictly prohibited.

3DO's LICENSOR(S) MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE SOFTWARE. 3DO'S LICENSOR(S) DOES NOT WARRANT, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME JURISDICTIONS. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

Other brand or product names are the trademarks or registered trademarks of their respective holders.

Contents

dump3do	This command lets you examine the contents of 3DO executables and object modules.	LC-3
link3do.....	This command lets you build an M2 executable or DLL from object files, DLL modules, and libraries.	LC-5

Assembly

__itof	Integer to float conversion.....	LC-11
__utof	Integer to float conversion.....	LC-11
DATAChunkify	A DataStream tool that prepares files for the DATA subscriber.	LC-15

Chapter 1

Dev_Commands

dump3do

This command lets you examine the contents of 3DO executables and object modules

Synopsis

```
dump3do
    [-ofname]
    [-?] [-V] [-h] [-p] [-s] [-t] [-r]
    [-d] [-a] [-g] [-c] [-u]
    [-l] [-i] [-b] [-m] objfile...
```

Description

dump3do prints information about one or more object files. The options control what particular information to display.

objfile... are the object files to be examined. When you specify archives, objdump shows information on each of the member object files.

Arguments

- m
If any files from objfile are archives, display the archive header information.
- u
Disassemble text section. Display the assembler mnemonics for the machine instructions from objfile. This option only disassembles those sections (.text sections) which are expected to contain instructions.
- h
File header. Display summary information from the overall header of each file in objfile.
- s
Section headers. Display summary information from the section headers of the object file.
- p
Program headers. Display summary information from the program headers of the object file.
- ? Help
Print a summary of the options to this program and exit.
- l
Label the display (using debugging information) with the filename and source line numbers corresponding to the object code shown. Only useful with -u.
- r
Relocation Entries. Print the relocation entries of the file.
- d
Dynamic table. Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.
- c
Section contents. Display the full contents of any sections.

- i Interpret symbolic info
- ofname Redirects output to output file
- t Symbol Table. Print the symbol table entries of the file.
- V Print the version number of this program and exit.
- a Hash table contents.
- g Debug info
- b Binary dump

Copying

The 602 disassembly portion of Dump3DO is based on code which is Copyright (c) 1991 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified version of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.

link3do

This command lets you build an M2 executable or DLL from object files, DLL modules, and libraries

Synopsis

```
link3do
  [-B[d=data_base,t=text_base,i=image_base]]
  [-e symbol] [-llname]
  [-m[2|fname]] [-ofname] [-r] [-i] [-s[s]]
  [-Lpath...] [-b[secname]=base] [-Aalignment]
  [-xdef_file] [-v] [-g] [-G] [-k] [-n] [-D] [-U]
  [-N] [-R] [-Hfield=val...] [-V] [-Mmagicno]
  objfile... [modulefile...] [-llname...]
```

Description

Link3DO reads various input files (libraries, dllmodules, and object files produced by the compiler or assembler), resolves and relocates references between and within them, and creates an m2 executable file or DLL module in 3DO ELF format.

objfile... are the object files. modulefile... are the DLL module files to be linked.

Arguments

```
-B[d=data_base,t=text_base,i=image_base]
  Set [data&bss|text|image] base

-esymbol
  Use "symbol" as entry point

-llname
  Use library lib"lname".a The form -lfullpath is also
  supported. The latter form does not prefix "lib" or
  append ".a"

-m[2|fname]
  Generate map file to stdout [fname]

-ofname
  Output file (default is a.out)

-r
  Generate relocations in file to keep file relative

-i
  Incremental link

-s[s]
  Strip unnessesary stuff from file [.comment too]

-Lpath
  Add this directory path to the list of paths searched for
  libraries.
  This affects libraries used with the -llname option.

-b[secname]=base
```

Set section base

-Aalignment
Set section alignment

-xdef_file
Use definitions file for resolving imports/exports

-v
Be verbose

-g
Generate debug info in file

-G
Generate debug info to external .sym file

-k
Keep everything in file

-n
Generate standard Elf file

-D
Allow duplicate symbols

-U
Allow undefined symbols

-Hfield=val
Set the field in the 3do header
Field names:

- Hname=<string> : sets the app name.
- Hpri=<n> : sets the app priority
- Hversion=<n> : sets the app version
- Hrevision=<n> : sets the app revision
- Htype=<n> : sets the app type
- Hsubsys=<n> : sets the app subsystem
- Htime=<date|now> : sets the app time ('MM/DD/YY HH:MM:SS' or 'now')
- Hosversion=<n> : sets the app osversion
- Hosrevision=<n> : sets the app osrevision
- Hfreespace=<n> : sets the app freespace
- Hmaxusecs=<n> : sets the app maxusecs
- Hflags=<n> : sets the app flags

-N
Do not autoload DLLs

-R
Allow all DLLs to be reloaded

-V
Prints the version of the linker.

-Mmagicno

Use "magicno" as the magic number.

Chapter 2

Misc

__itof __utof

Integer to float conversion

Synopsis

```
float __itof( int i )  
float __utof( unsigned int i )
```

Description

This function is used by the compiler and assembly routines to convert an integer value to a float.

Arguments

An integer in register r3

Return Value

Returns a float in register f1

Implementation

Assembly function in libc.

Caveats

Destroys contents of registers r3-r6 and f1. Requires a valid SP.

Chapter 3

Tools

DATAChunkify

A DataStream tool that prepares files for the DATA subscriber.

Format

```
DATAChunkify <flags>
  -i      <file>      input file name [REQUIRED].
  -o      <file>      output file name [REQUIRED].
  -chan   <num>       channel number (0).
  -t      <ticks>     first chunk time in audio folio ticks
(0).
  -to     <ticks>     time offset for subsequent chunks (0).
  -cs     <size>       size of a data chunk in BYTES (16384).
  -m      <num>       memtype bits (0).
  -u      <type><type> user data word (0000000000000000).
  -comp   <type>      compressor (NONE).
```

Valid compressors are:

```
NONE    No compression.
3DOC    3DO compression folio compression.
-? or -help for usage info.
```

Description

This tool breaks files into chunks suitable for the DATA subscriber. The files output by this tool must be woven into a stream file with the Weaver tool in order to be used by the DataStreamer.

